Efficient distributed reachability querying of massive temporal graphs

Zhang, Tianming; Gao, Yunjun; Chen, Lu; Guo, Wei; Pu, Shiliang; Zheng, Baihua; Jensen, Christian S.

Published in:
VLDB Journal

DOI (link to publication from Publisher):
10.1007/s00778-019-00572-x

Publication date:
2019

Document Version
Accepted author manuscript, peer reviewed version

The VLDB Journal manuscript No.
(will be inserted by the editor)

Efficient Distributed Reachability Querying of Massive Temporal Graphs

Tianming Zhang · Yunjun Gao · Lu Chen · Wei Guo · Shiliang Pu · Baihua Zheng · Christian S. Jensen

Received: 9 November 2018 / Accepted: 17 September 2019

Abstract Reachability computation is a fundamental graph functionality with a wide range of applications. In spite of this, little work has as yet been done on efficient reachability queries over temporal graphs, which are used extensively to model time-varying networks, such as communication networks, social networks, and transportation schedule networks. Moreover, we are faced with increasingly large real-world temporal networks that may be distributed across multiple data centers. This state of affairs motivates the paper's study of efficient reachability queries on distributed temporal graphs. We propose an efficient index, called *Temporal Vertex Labeling* (*TVL*), which is a labeling scheme for distributed temporal graphs. We also present algorithms that exploit *TVL* to achieve efficient support for distributed reachability querying over temporal graphs in Pregel-like systems. The algorithms exploit several optimizations that hinge upon non-trivial lemmas. Extensive experiments using massive real and synthetic temporal graphs are conducted to provide detailed insight into the efficiency and scalability of the proposed methods, covering both index construction and query processing. Compared with the state-of-the-art methods, the *TVL* based query algorithms are capable of up to an order of magnitude speedup with lower index construction overhead.

Keywords Graph · Reachability · Distributed processing · Query processing · Algorithm

Tianming Zhang · Yunjun Gao (Corresponding Author) · Wei Guo
College of Computer Science
Zhejiang University, Hangzhou, China
E-mail: {tianmingzhang, gaoyj, weiguo}@zju.edu.cn

Yunjun Gao
The Key Lab of Big Data Intelligent Computing of Zhejiang Province, Zhejiang University, Hangzhou, China
E-mail: gaoyj@zju.edu.cn

Lu Chen · Christian S. Jensen
Department of Computer Science
Aalborg University, Aalborg, Denmark
E-mail: {luchen, csj}@cs.aau.dk

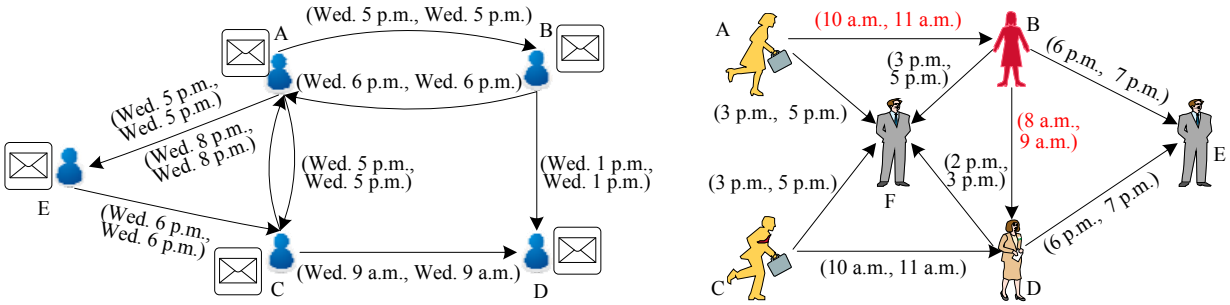
Shiliang Pu
Hangzhou Hikvision Digital Technology Co., Ltd.
Hangzhou 310052, China
E-mail: pushiliang@hikvision.com

Baihua Zheng
School of Information Systems
Singapore Management University, Singapore
E-mail: bhzheng@smu.edu.sg

1 Introduction

Graphs are often employed to represent relationships between entities. The literature contains many studies of general graphs [9], where no temporal dimension is considered. Yet, temporal information is important in existing and emerging real-life applications, where the relationships between entries are intermittent (i.e., a relationship is established at a specific moment and persists only for some time). Such relationships can be captured by temporal graphs (e.g., transportation schedule networks, telephone or email networks, social networks). This paper concerns efficient temporal-graph *reachability querying*, which constitutes fundamental graph functionality, and has many important applications such as path computation, query processing, and graph analysis and mining. In the following, we consider two representative examples.

Example 1 (**Email network**). In an email network, vertices represent senders and recipients of emails. A



(a) Email network (b) Social network

Fig. 1 Example of temporal networks

directed temporal edge from vertex u to vertex v, denoted as (u, v, s_t, a_t), indicates that u sends v an email at time s_t and that v receives the email at time a_t. Then, an email transaction has a duration $I = a_t - s_t$. It is worth mentioning that, email exchanges in most email networks are instantaneous, i.e., $s_t = a_t$ and $I = 0$. Fig. 1(a) illustrates an example. For each edge, we omit u and v but represent s_t and a_t in the form of (s_t, a_t). In the email network, reachability queries can be employed to determine how a group of persons share information over time, or study how computer viruses spread through a community.

Example 2 (**Social network**). In a social network, each vertex denotes a person. Temporal edges can represent multiple types of interactions. For instance, a directed temporal edge $e = (u, v, s_t, a_t)$ might mean that u initiates a face-to-face meeting with v from time s_t to time a_t, indicating that the interaction lasts for a period of time $I = a_t - s_t$. Fig. 1(b) shows a simple network. Reachability queries in social networks can help people understand how information flows or how opinions form over time.

Although reachability queries have been investigated extensively, there remain challenges. First, most existing studies are designed for general graphs and do not extend to temporal graphs, where traversals are not transitive. In the above two examples, if we disregard the temporal information and treat the graphs as general graphs, results of reachability queries can be misleading. In Example 2, suppose A is infected with influenza and a doctor wants to determine whether D is potentially infected by A in order to decide whether D should be quarantined. If we ignore the temporal information, we conclude that D is potentially infected by A since there is a path from A to D via B. However, D cannot be infected by A because the interaction between A and B (from 10 a.m. to 11 a.m.) only occurs after B interacts with D (from 8 a.m. to 9 a.m.), meaning that the virus carried by A cannot have been transmitted to D. Second, as the scale of real-world temporal graphs keeps increasing rapidly (e.g., the number of phone calls recorded each day), it is increasingly important to study how the operations on graphs can be distributed across data centers for higher efficiency and scalability. To this end, we aim to enable efficient distributed reachability queries over temporal graphs.

To handle reachability queries on temporal graphs, a straightforward approach is to perform breadth-first search (BFS) directly, which has time complexity $O(|V| + |E|)$, where $|V|$ and $|E|$ are the number of vertices and edges, respectively. This is prohibitive for massive graphs. Considering that reachability queries over general graphs can be supported by existing approaches [8, 11, 47, 48], another naïve solution is to transform temporal graphs into general graphs without loss of reachability information, and then apply existing indexing methods (such as the state-of-the-art distributed approaches DRQ [8] and DSR [11]) to answer reachability queries. As a representative example, $TopChain$ [41] first transforms a temporal graph into a directed acyclic graph (DAG) and then makes use of properties of temporal graphs to design indexes and query algorithms on the transformed graph. Nonetheless, a transformed graph can be tens of times larger and much denser than the original temporal graph, incurring very high index construction overhead and poor query performance, as will be seen in our experiments.

We propose a new indexing technique called TVL that is a labeling scheme. The basic idea is to maintain two sets of temporal labels for each vertex v, namely $Lin(v)$ and $Lout(v)$. $Lin(v)$ records the vertices u that can reach v, together with the time starting at u and the time ending at v. $Lout(v)$ records the vertices w that v can reach, together with the time starting at v and the time ending at w. To reduce the index size and to construct the index efficiently, TVL only maintains *canonical temporal labels* instead of all temporal

labels, and it adopts a message propagation technique to efficiently construct the index *without graph transformation*. Several query processing algorithms are developed on top of *TVL* to support distributed reachability queries on temporal graphs, and a suite of non-trivial lemmas state properties that are exploited to improve query performance. As a result, compared with existing techniques, *TVL* has much lower query cost and lower index construction overhead.

In a nutshell, the key contributions are as follows.

- We present a new scalable index structure, *TVL*, which does not rely on graph transformation and that can be constructed efficiently by adopting message propagation techniques. Furthermore, *TVL* supports efficient insertion operations.
- We provide several search algorithms using *TVL* to answer distributed reachability queries on temporal graphs, and we develop several non-trivial lemmas that enable improved performance.
- We report extensive experiments using seven real and synthetic datasets that offer detailed insight into the efficiency and scalability of the proposed techniques. Compared with the state-of-the-art distributed reachability methods [8,11,41], *TVL* is from several times to an order of magnitude faster in terms of query efficiency, with much smaller index size and lower index construction cost.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 formalizes the paper's problem. Section 4 details baselines. Section 5 covers the *TVL* based method. Section 6 reports experimental results and our findings. Finally, Section 7 concludes the paper and offers directions for future work.

2 Related work

In this section, we review previous related studies on *reachability queries on general graphs*, *temporal graphs*, and *distributed graph processing systems*.

2.1 Reachability queries on general graphs

Existing efforts on reachability queries on general graphs in the literature [1,4,5,6,15,16,17,18,19,29,30,32,34,35,36,38,46,47,48,51,52] mostly focus on developing centralized algorithms, and only a few studies [8,11] aim at designing distributed algorithms. Cheng et al. [49] offer a comprehensive survey on reachability querying. In summary, existing efforts are all devoted to attaining less online reachability query time as well as constructing an offline index that can significantly reduce the space consumption. Specifically, approaches to reachability queries mainly fall into three categories: *transitive closure based querying*, *hop labeling based retrieval*, and *depth-first based search with pruning*.

Given a graph $G = (V, E)$, the transitive closure of a vertex $v \in V$ consists of the set of vertices in G that can be reached from v. Transitive closure based querying methods [1,5,15,17,19,29,36] first precompute the transitive closures of all vertices in G. Then, for a reachability query that asks whether a source vertex s can reach a target vertex t in G, those methods first retrieve the transitive closure of s and then check whether t is contained in the transitive closure of s. It is observed that a query can be answered in $O(1)$ time, while precomputing the transitive closures of all vertices takes $O(|V||E|)$ time. Thus, transitive closure based querying methods trade index construction and storage overhead for query efficiency.

Hop labeling based retrival methods [6,16,18,46,52] precompute an offline index by constructing an out-label set $Lout(v)$ and an in-label set $Lin(v)$ for each vertex v in G. $Lout(v)$ records a list of intermediate vertices that v can reach, and $Lin(v)$ records the set of vertices that can reach v. Then, a reachability query can be answered by finding a common vertex in the intersection of the source vertex's out-label set and the target vertex's in-label set. Overall, *hop labeling based retrival* methods can answer reachability queries with high query efficiency when label sets are small.

Depth-first based search with pruning methods [4,30,32,34,38,47,48] are designed to reduce the offline index construction time. The basic idea is to utilize depth-first search (*DFS*) on a graph G to answer reachability queries and to rely on auxiliary labeling information to prune the search space. The index construction time and index size of this class of methods are both small, enabling these methods to scale to sizable graphs.

Since centralized approaches are limited to the main memory of a single machine, a few techniques aim to enable distributed reachability querying. We are aware of two techniques [8,11] that specifically tackle the problem of distributed reachability querying. One [8] is for single-source, single-target reachability querying, and the other [11] is for set reachability querying, which is a generalized form of reachability queries.

Fan et al. [8] propose a distributed algorithm called *disReach*. It first precomputes the local reachability between in-boundaries and out-boundaries in graph partitions in parallel. A reachability query can then be answered by *DFS* on a dependency graph that is constructed based on the precomputed reachability information. Gurajada and Theobald [11] develop a graph-based index structure called *compound graph*, which is

obtained by merging local partitions with partition-specific boundary graphs. Based on the precomputed compound graphs, reachability queries can be answered locally or by using a single step of message exchange.

It is worth mentioning that, the aforementioned algorithms are designed for general graphs and cannot be applied to temporal graphs directly, since temporal information is ignored and traversal in temporal graph is not transitive. If temporal graphs are transformed into general graphs, existing distributed approaches can be used. Nonetheless, they are inefficient, as to be verified in our experiments.

In this paper, we propose *TVL* index, which is more scalable than 2-hop labeling. As pointed out in GRAIL [47], 2-hop labeling suffers from poor scalability, especially for large graphs. Consequently, we do take scalability into consideration when we design *TVL*. For example, in order to reduce the storage overhead of 2-hop labeling, *TVL* introduces the new concepts of *vertex significance value* and *canonical temporal labels*, and sorts the canonical temporal labels of *TVL* in ascending order of vertex significance values. It strategically stores canonical temporal labels with the top-k smallest significance values to effectively reduce the index storage overhead. As to be verified in the scalability experiments of *TVL*, *TVL* has smaller construction time and is more scalable than *Grail*, while the construction cost and the storage size complexities of *Grail* are theoretically less than 2-hop labeling.

2.2 Temporal graphs

Temporal graphs have garnered substantial research interest. Several comprehensive surveys [3,12,20] are available in the literature.

Existing studies on temporal graphs are related to topics such as connected components [25], temporal graph traversals [13], subgraph isomorphism [27], temporal paths [26,37,39], temporal subgraph mining [45], minimum spanning trees [14], and the traveling salesman problem [24]. These studies generally aim at theoretical analysis of concepts, formalisms, models, and metrics for temporal graphs.

Recently, *TopChain* [41], a centralized algorithm for reachability queries on temporal graphs was proposed. The algorithm first transforms the original temporal graph into a directed acyclic graph (*DAG*) and then decomposes the *DAG* into a set of ranked chains. Next, based on the top-k chains, for every vertex v in the *DAG*, *TopChain* constructs two sets of labels, $Lin(v)$ and $Lout(v)$, which record the last and first vertices in a top-k chain that can reach v and that is reachable from v, respectively. Reachability queries can be answered using the vertex labels together with *DFS*. We extend this algorithm to a distributed algorithm called *PTopChain*, implemented in Blogel, in order to support distributed reachability queries over massive temporal graphs. It turns out that *PTopChain* has high index construction overhead. This is because the transformed graphs are much larger than the original temporal graphs. To reduce index construction cost and speed up online search, we present a novel indexing technique called *Temporal Vertex Labeling* (*TVL*) that offers better query performance with smaller index size, to covered in detail in experiments.

2.3 Distributed graph processing systems

Batarfi et al. [2] provide a comprehensive survey of state-of-the-art distributed graph processing platforms. Representatives include MapReduce [7], Pregel [23], Giraph++ [33], Blogel [42], GraphLab [22], Trinity [31], Spark [50], and GraphX [10].

MapReduce [7] is adopted by corporations for big data processing. Nevertheless, it is ill-suited for iterative algorithms. In contrast, Pregel-like systems, which include also Giraph++ and Blogel, are proposed for supporting iterative graph computations. Pregel [23], which is based on a bulk synchronous parallel model, is introduced by Google for graph applications. Apache Giraph[1] and Apache Hama[2] are open-source implementations of Pregel. Giraph++ [33], which is built on top of Giraph, represents a shift from a node-centric to a graph-centric computing system. Blogel [42], which is implemented in C++, provides vertex-centric, block-centric, and global interfaces for programming algorithms. It supports three types of jobs: (i) vertex-centric graph computing; (ii) graph partitioning; and (iii) block-centric graph computing. GraphLab [22] is an open-source project that encompasses a family of related systems. Trinity [31] and Spark [50] are memory-based distributed processing systems. GraphX [10] is built on Spark for graph-parallel computation.

We design our algorithms within the setting of the Pregel-like systems [44], because those systems are best suited for iterative graph query processing. Pregel-like systems first distribute vertices of the input graph to different machines. To perform computational tasks, a user-defined function *compute*() needs to be implemented. Then, the tasks proceed as sequences of supersteps. In every superstep, each active vertex invokes *compute*() to perform a user-specified task. A task ter-

[1] Giraph is available at http://giraph.apache.org/.
[2] Hama is available at http://hama.apache.org/.

Table 1 Symbols and description

Annotation	Description
$G = (V, E)$	a temporal graph with a set V of vertices and a set E of temporal edges (see Definition 1)
$\mathscr{G} = (\mathscr{V}, \mathscr{E})$	the transformed directed acyclic graph of G
$c = \langle v_1, v_2, \cdots, v_m \rangle$	a chain that is an ordered sequence of m reachable vertices
$\mathbb{C} = \{c_1, c_2, \cdots, c_l\}$	a chain cover of $\mathscr{G}$, which is a disjoint partition of $\mathscr{V}$
$\|V\|$ or $\|E\|$	the number of vertices or edges
u,v,w,s or t	a vertex
s_t,a_t,w_s or w_a	a timestamp
(u, v, s_{t_i}, a_{t_i}) or e	a temporal edge
$p(u, v)$	a time-respecting path from u to v (see Definition 2)
S_p, E_p, D_p, d_p	the start time, end time, duration, and length of a path p, respectively
$(minV, sig, s_t, a_t)$	a temporal in-label or out-label of a vertex (see Definition 7 or Definition 8)
$N_{in}(u)$ or $N_{out}(u)$	the set of u's in-neighbors or out-neighbors
$D_{in}(u)$ or $D_{out}(u)$	the in-degree or out-degree of a vertex u
$n(v)$	the topological level number of a vertex v
$\rho(v)$	the significance value $\rho(v)$ of a vertex v
k	a parameter for controlling the size of an index
$TRP_G\ (s, t, I = [w_s, w_a])$	a function that returns all time-respecting paths p from s to t such that $[S_p, E_p] \subseteq I$
$L_{in}(v)$ or $L_{out}(v)$	the in-label or out-label set of a vertex v
$TRQ(s, t, I)$	a temporal reachability query (see Definition 3)
$EETQ(s, t, I)$	an earliest ending time query (see Definition 4)
$MDQ(s, t, I)$	a minimum duration query (see Definition 5)
$EXTRACT(C, x, con)$	a function that extracts all the labels la in a given label set C such that $la.minV = x$ and Boolean condition con is true

minates when all vertices vote to halt, and there is no message in transmit. In addition, Pregel-like systems support message combiners. Users can implement a *combine*() function to combine messages sent to the same vertex, thus reducing the number of messages to be buffered and transmitted. Pregel-like systems also support aggregators that can be used for capturing the global state of the graph.

In this paper, we implement our approaches on the popular Pregel-like system Blogel [42], which has been shown to be more efficient than other Pregel-like systems (such as Giraph and Giraph++). The main reason is that Blogel is implemented in C++. In addition, we choose Blogel [42] instead of Pregel+ [43]. This is because, Blogel has VB (Vertex&Block) computing model and supports block level communication, which is more suitable for reachability queries [42]. Moreover, it is also proved by the experimental comparison that the PageRank on Blogel (shown in Fig. 16 of Blogel [42]) is faster than that on Pregel+ (depicted in Fig. 12 of Pregel+ [43]) on the same dataset WebUK.

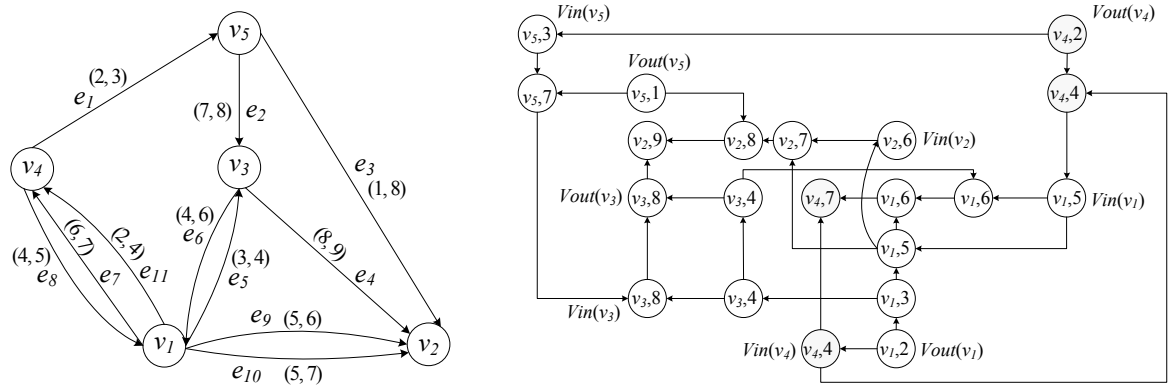
3 Problem formulation

We first define the notions of *temporal graph* and *time-respecting path*. Following Wu et al. [41], we assume that graphs are directed; and an undirected edge can be modeled by two directed edges. For ease of reference, Table 1 summarizes notations used frequently in this paper.

Definition 1 (Temporal Graph). A temporal graph $G = (V, E)$ consists of a set V of vertices and a set E of temporal edges. In particular, a temporal edge $e_i \in E$ from a vertex $u \in V$ to another vertex $v \in V - \{u\}$, in the form of a quadruple (u, v, s_{t_i}, a_{t_i}), indicates that an event from u to v starts at time s_{t_i} and ends at time a_{t_i}, thus having duration $I = a_{t_i} - s_{t_i}$.

Definition 2 (Time-respecting Path) [28]. A time-respecting path p from u to v, denoted by $p(u, v) = \langle u, e_1, w_1, \cdots, w_{m-1}, e_m, v \rangle$, is defined as a sequence of contacts with non-decreasing times, where $\langle u, w_1, \cdots, w_{m-1}, v \rangle$ and $\langle e_1, e_2, \cdots, e_m \rangle$ are sequences of vertices and temporal edges, respectively, with $e_1 = (u, w_1, s_{t_1}, a_{t_1})$, $e_m = (w_{m-1}, v, s_{t_m}, a_{t_m})$, and $e_i = (w_{i-1}, w_i, s_{t_i}, a_{t_i})$ for $i \in (1, m)$, such that, for any $i \in [1, m)$, $a_{t_i} \leq s_{t_{i+1}}$. We refer to $S_p = s_{t_1}$ as the start time of p, and $E_p = a_{t_m}$ as the end time of p. Further, we refer to $D_p = a_{t_m} - s_{t_1}$ as the duration of p, and to $d_p = m$ (i.e., the number of the edges in p) as the length of p.

Based on the definition of a temporal graph G and a time-respecting path p, we introduce a function $TRP_G(s, t, I{=}[w_s, w_a])$ on a temporal graph G that returns all directed time-respecting paths p from a vertex s to another vertex t such that $S_p \geq w_s$ and $E_p \leq w_a$. Then, given a temporal graph G, a source vertex s, a target vertex t, and a time interval $I{=}[w_s, w_a]$, where w_s and w_a are the user-specified start time and end time of a query, we define three types of queries on G, namely,



(a) A temporal graph G (b) Transformed graph $\mathscr{G}$ of G

Fig. 2 Example of a temporal graph G, and G's transformed graph $\mathscr{G}$

temporal reachability query, *earliest end time query*, and *minimum duration query*.

Definition 3 (Temporal Reachability Query). A temporal reachability query from s to t within I on G, denoted as $TRQ(s, t, I)$, returns true if $TRP_G(s,\ t,\ I)$ finds at least one path; otherwise, it returns false.

Definition 4 (Earliest End Time Query). An earliest end time query from s to t within I on G, denoted as $EETQ(s, t, I)$, retrieves the earliest end time $min(E_p)$ among all paths p returned by $TRP_G(s,\ t,\ I)$.

Definition 5 (Minimum Duration Query). A minimum duration query from s to t within I on G, denoted as $MDQ(s, t, I)$, finds the minimum duration $min(D_p)$ among all paths p returned by $TRP_G(s,\ t,\ I)$.

Example 3 Fig. 2(a) shows a temporal graph G, where $V = \{v_1, v_2, \cdots, v_5\}$ and $E = \{e_1, e_2, \cdots, e_{11}\}$. A tuple of the form (s_t, a_t) associated with each temporal edge e indicates the start and end times of e. The temporal reachability query $TRQ(v_4,\ v_3,\ [1, 4])$ returns false since no time-respecting path exists that connects v_4 to v_3 in interval $[1, 4]$, i.e., $TRP_G(v_4,\ v_3,\ [1, 4]) = \emptyset$. On the other hand, $TRP_G(v_5,\ v_2,\ [1, 10]) = \{p_1,\ p_2\}$ with $p_1 = \langle v_5,\ e_2,\ v_3,\ e_4,\ v_2\rangle$ and $p_2 = \langle v_5,\ e_3,\ v_2\rangle$. Therefore, the earliest end time query $EETQ(v_5,\ v_2,\ [1, 10])$ returns $min(E_{p_1},\ E_{p_2}) = 8$, and the minimum duration query $MDQ(v_5,\ v_2,\ [1, 10])$ returns $min(D_{p_1},\ D_{p_2}) = 2$.

4 Baseline methods

To compute distributed reachability queries on massive temporal graphs, a simple approach is to perform a distributed breadth-first search (BFS) on the temporal graph starting from the source vertex and continuing until either the target vertex is reached or it is determined that no such time-respecting path exists. This approach requires no index, but requires $O(|V| + |E|)$ time for each query, which is prohibitively expensive for massive graphs.

Another naïve approach is to transform temporal graphs into general graphs without loss of reachability information, and then apply existing indexing methods to answer reachability queries. The state-of-the-art centralized method *TopChain* [41] is based on this idea. *TopChain* first transforms the temporal graph G into a directed acyclic graph (DAG) $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ and then decomposes the transformed graph $\mathscr{G}$ into a set of chains (i.e., a chain cover $\mathbb{C}$). For each chain c in $\mathbb{C}$, *TopChain* assigns a unique rank randomly or in descending order of the degrees of all the vertices in the chain c. Then, *TopChain* computes an in-label set $Lin(v)$ and an out-label set $Lout(v)$ for each vertex $v \in \mathscr{V}$. $Lin(v)$ and $Lout(v)$ maintain the last and first vertex in the top-k smallest ranking chains that can reach v and that is reachable from v, respectively. Here, a chain $c = \langle v_1, v_2, \cdots, v_m\rangle$ is an ordered sequence of m reachable vertices such that v_i can reach v_{i+1} for $1 \le i < m$. A chain cover $\mathbb{C} = \{c_1, c_2, \cdots, c_l\}$ of $\mathscr{G}$ is a disjoint partition of $\mathscr{V}$, where c_i is a chain for $1 \le i \le l$. Specifically, the main steps of constructing *TopChain* index are summarized as follows.

First, a temporal graph $G = (V, E)$ is transformed into a general graph $\mathscr{G} = (\mathscr{V}, \mathscr{E})$, which is proven to be a DAG in the full version [40] of the paper [41]. The detailed transformation is provided below.

Vertex transformation. Each vertex $w \in V$ is transformed into two sets of vertices (i.e., $Vin(w)$ and $Vout(w)$) in $\mathscr{V}$, where $Vin(w) = \{\langle w, a_{t_i}\rangle \mid 1 \le i \le h\}$ and $Vout(w) = \{\langle w, s_{t_j}\rangle \mid 1 \le j \le m\}$. Here, a_{t_i} is a distinct arrival time instance at which edges from in-neighbors of w arrive at w; and s_{t_j} is a distinct start

Algorithm 1: PTCA Algorithm

```
Input: a transformed graph 𝒢 = (𝒱, ℰ), a topological
       level number n, an integer k, a chain cover
       ℂ={c_1, ··· , c_l} of 𝒢 where c_i is ranked before
       c_j for 1 ≤ i < j ≤ l
Output: Lin(v) for every vertex v ∈ 𝒱
1:  foreach vertex v ∈ 𝒱 do
2:      if superstep = 0 then
3:          Lin(v) ← {(v.x, v.y)}
4:          if n(v) = 0 then
5:              foreach out-neighbor w ∈ N_out(v) do
6:                  send message Lin(v) to w
7:      else if there is no message in transmit then
8:          return Lin(v)
9:      else
10:         foreach message Lin(u) sent from u ∈ N_in(v)
            do
11:             L ← Lin(v) ∪ Lin(u)
12:         Let L_k be the top k labels with the smallest
            chain rank from L such that if ∃(u.x,u.y)∈L
            and (w.x,w.y)∈L satisfies u.x = w.x and
            u.y<w.y, i.e., u and w are in the same chain,
            then (u.x, u.y)∉L_k
13:         Lin(v) ← L_k
14:         if n(v) = superstep then
15:             foreach out-neighbor w ∈ N_out(v) do
16:                 send message Lin(v) to w
```

time instance at which an edge starts from w to its out-neighbors. h and m are the numbers of distinct arrival time instances and start time instances, respectively.

Edge transformation. The edge transformation involves three steps. (i) Each edge e $(= (u, w, s_t, a_t)) \in E$ is transformed into an edge from the vertex $\langle u, s_t\rangle \in Vout(u)$ to the vertex $\langle w, a_t\rangle \in Vin(w)$. (ii) Given $Vin(w) = \{\langle w, a_{t_1}\rangle, \langle w, a_{t_2}\rangle, \cdots, \langle w, a_{t_h}\rangle\}$, where $a_{t_i} < a_{t_{i+1}}$ for $1 \le i \le (h-1)$, a new directed edge from vertex $\langle w, a_{t_i}\rangle \in Vin(w)$ to vertex $\langle w, a_{t_{i+1}}\rangle \in Vin(w)$ is created in $\mathscr{E}$. For $Vout(w)$, the edges are created in the same way. (iii) For each vertex $\langle w, a_{t_{in}}\rangle \in Vin(w)$, according to its reverse order in $Vin(w)$, a directed edge from vertex $\langle w, a_{t_{in}}\rangle$ to vertex $\langle w, s_{t_{out}}\rangle \in Vout(w)$ is created, where $s_{t_{out}} = min\{s_{t'} \mid \langle w, s_{t'}\rangle \in Vout(w), s_{t'} \ge a_{t_{in}}\}$, and no edges from other vertices $\langle w, a_{t'_{in}}\rangle \in Vin(w)$ to $\langle w, s_{t_{out}}\rangle$ have been created.

Example 4 Fig. 2(b) shows the transformed graph $\mathscr{G}$ of the temporal graph G depicted in Fig. 2(a). For instance, v_4 is transformed into $Vin(v_4) = \{\langle v_4, 4\rangle, \langle v_4, 7\rangle\}$ and $Vout(v_4) = \{\langle v_4, 2\rangle, \langle v_4, 4\rangle\}$, and $(v_4, v_5, 2, 3)$ in E is transformed into an edge from $\langle v_4, 2\rangle \in Vout(v_4)$ to $\langle v_5, 3\rangle \in Vin(v_5)$ in $\mathscr{E}$. For $Vout(v_4)$, an edge from $\langle v_4, 2\rangle$ to $\langle v_4, 4\rangle$ is created in $\mathscr{E}$. For $Vin(v_4)$, an edge from $\langle v_4, 4\rangle$ to $\langle v_4, 7\rangle$ is created in $\mathscr{E}$. Moreover, an edge from $\langle v_4, 4\rangle \in Vin(v_4)$ to $\langle v_4, 4\rangle \in Vout(v_4)$ is created in $\mathscr{E}$. Note that, $\langle v_4, 4\rangle \in Vin(v_4)$ and $\langle v_4, 4\rangle \in Vout(v_4)$ are different vertices.

In the rest of this section, we use v instead of $\langle w, s_t\rangle$ to represent a vertex in $\mathscr{G}$ for brevity.

Second, the transformed graph $\mathscr{G}$ is decomposed into a set $\mathbb{C}$ of ranked chains (i.e., the chain cover of $\mathscr{G}$), and each vertex $v \in \mathscr{V}$ is assigned a chain code $(v.x, v.y)$. Here, a chain $c \in \mathbb{C}$ is an ordered sequence of reachable vertices, and $v.x$ is the rank of chain c and $v.y$ is the position of v in c. Note that, during the process of graph transformation, each set of vertices $Vin(w)$ or $Vout(w)$ appears as a chain due to the properties of temporal graphs; thus, a natural chain cover $\mathbb{C} = \{Vin(w) \mid w \in V\} \cup \{Vout(w) \mid w \in V\}$ of $\mathscr{G}$ and chain code for every vertex are obtained.

Finally, based on the transformed graph $\mathscr{G}$, the chain cover, and the chain codes of all vertices, $Lin(v)$ and $Lout(v)$ for each vertex $v \in \mathscr{V}$ are computed. $Lin(v)$ keeps the last vertex in top-k chains that can reach v, and $Lout(v)$ keeps the first vertex in top-k chains that is reachable from v. Here, chains are ranked in descending order of their ϕ values, which is the sum of out-edges and in-edges of all vertices in a chain. The top-k chains are those having top-k smallest chain ranks.

We parallelize *TopChain* to achieve *PTopChain*. To enable parallelism, *PTopChain* is constructed using the topological level number [6] of a vertex rather than the topological order. The topological level number of a vertex v, denoted as $n(v)$, is defined below.

$$n(v) = \begin{cases} 0 & N_{in}(v) = \emptyset \\ max_{u \in N_{in}(v)}(n(u) + 1) & otherwise \end{cases}$$

Here, $N_{in}(v) = \{u \mid (u, v) \in \mathscr{E}\}$ denotes the set of v's in-neighbors. The topological level number $n(v)$ can be calculated iteratively. To accelerate the computation, each vertex first caches incoming messages, and then, it computes the topological level number and sends messages to neighbors until all needed messages are received. Given a transformed graph $\mathscr{G} = (\mathscr{V}, \mathscr{E})$, a topological level number n, an integer k, and a chain cover $\mathbb{C} = \{c_1, \cdots, c_l\}$ of $\mathscr{G}$, a *PTopChain Construction Algorithm* (PTCA) is developed. The pseudo-code is shown in Algorithm 1.

Initially, graph $\mathscr{G}$ is partitioned across a cluster of workers; then computation tasks including a consecutive of supersteps are performed iteratively. Let $N_{out}(v) = \{w \mid (v, w) \in \mathscr{E}\}$ be the set of v's out-neighbors and $N_{in}(v) = \{u \mid (u, v) \in \mathscr{E}\}$ be the set of v's in-neighbors. First, we consider superstep $= 0$. For every vertex $v \in \mathscr{V}$, PTCA assigns the chain code of each vertex v to $Lin(v)$ (line 3). Then, if $n(v) = 0$, PTCA sends $Lin(v)$ to its out-neighbor $w \in N_{out}(v)$ (lines 4–6). Next, if there is

no message to transmit, the in-label sets of all vertices are computed, and PTCA terminates (lines 7–8). Otherwise, for the vertex v that receives $Lin(u)$ from its in-neighbor $u \in N_{in}(v)$, PTCA computes $Lin(v)$. $Lin(v)$ keeps the top k labels with the smallest chain rank from $Lin(u)$ and the in-labels of v such that, for each chain, $Lin(v)$ only contains the chain code of the last vertex that can reach v (lines 9–13). If $n(v)$ = superstep, meaning that $Lin(v)$ has been computed, PTCA sends $Lin(v)$ to its out-neighbors for computing labels in the following supersteps (lines 14–16).

$Lout(v)$ can be computed similarly, and hence, we omit its pseudo-code. The only difference between $Lout(v)$ computation and $Lin(v)$ computation is that PTCA needs to take the reverse graph of $\mathscr{G}$ as an input and that $n(v)$ is calculated by topological sorting of the reverse graph of $\mathscr{G}$. Algorithm *PTopChain* can be used for computing the three types of reachability queries defined in Section 3. The detailed algorithm is similar to Algorithm 2 presented by Wu et al. [41]. We note that *PTopChain* based algorithms process all traversed vertices in parallel, which accelerates search.

Discussion. Although *PTopChain* performs better than bi-directional *BFS* and other existing indexing methods in most cases, it still has shortcomings that limit its efficiency.

First, *PTopChain* transforms an original temporal graph into a *DAG*, which is probably tens of times larger than the original temporal graph, incurring extra index construction cost. Second, as *PTopChain* is a depth-first based search with pruning method, it performs *DFS* on the transformed *DAG* to answer reachability queries when index checks fail. This results in long query time since the transformed *DAG* is much larger than the original temporal graph.

To reduce the index construction overhead and improve the query efficiency, we propose a new index structure, called *Temporal Vertex Labeling* (*TVL*), to be detailed in the next section.

5 *TVL* based method

We first introduce a new index structure called *Temporal Vertex Labeling* (*TVL*). Compared with *PTopChain*, *TVL* has lower index construction cost and is much more compact. Then, we present the index construction algorithm and insertion scheme for *TVL*, and we cover several query algorithms using *TVL* that aim to efficiently support distributed reachability queries on temporal graphs. In the following, we say that a vertex u can *reach* another vertex v iff $\exists I = [w_s, w_a]$ with $w_s \leq w_a$ such that $TRP_G(u, v, I) \neq \emptyset$.

5.1 *TVL*

Baseline methods expand an original temporal graph into a much larger directed acyclic graph (*DAG*), which adversely affects query performance. In contrast, the *TVL* index is built directly on an original temporal graph without any graph transformation.

The basic idea of *TVL* is inspired by the typical 2-hop labeling scheme. Formally, both *TVL* and 2-hop labeling maintain two sets, $Lout(v)$ and $Lin(v)$, for each vertex v. However, they are different, and *TVL* is more scalable. First, 2-hop labeling is a *complete* index, which means that, for any vertex pair (u, v), u can reach v if and only if $Lout(u) \cap Lin(v) \neq \emptyset$. In other words, any reachability query can be simply answered by taking the intersection of the source vertex's out-label set and the target vertex's in-label set. This leads to high query efficiency when the label sets are small. Nonetheless, 2-hop labeling is unable to scale to massive graphs as label sets are often too large.

In contrast, *TVL* uses 2-hop labeling that attaches temporal information. In particular, *TVL* maintains two sets of temporal labels for every vertex $v \in V$, i.e., $Lin(v)$ and $Lout(v)$. Each temporal label in $Lin(v)$ records the vertices u that can reach v, together with the time starting at u and the time ending at v. Similarly, each temporal label in $Lout(v)$ records the vertices w that v can reach, together with the time starting at v and the time ending at w. However, storing all temporal labels in $Lin(v)$ and $Lout(v)$ is not practical in the case of massive graphs. Therefore, we use a parameter k and define canonical temporal labels (to be formalized in Definition 9) to control *TVL* size. We also define a vertex significance function ρ that ensures temporal labels in *TVL* maintain important vertices. *TVL* belongs to the category of methods that use depth-first based search with pruning. It incurs a small overhead in preprocessing, offers better query performance at a smaller index size, and is able to scale to large graphs.

Before we explain how to define ρ, we give definitions of the out-degree $D_{out}(u)$ and in-degree $N_{in}(u)$ of a vertex u in a temporal graph.

Definition 6 Given a temporal graph $G = (V, E)$, there may be multiple temporal edges from a vertex u to another vertex v. Let $S(u, v)$ be the set of temporal edges from u to v, and $|S(u, v)|$ be the cardinality. Then, the out-degree of u is denoted by $D_{out}(u) = \sum_{v \in N_{out}(u)} |S(u, v)|$, where $N_{out}(u) = \{v \mid (u, v, s_{t_i}, a_{t_i}) \in E\}$. The in-degree of u is defined as $D_{in}(u) = \sum_{w \in N_{in}(u)} |S(w, u)|$, where $N_{in}(u) = \{w \mid (w, u, s_{t_i}, a_{t_i}) \in E\}$.

Take the temporal graph G depicted in Fig. 2(a) as an example. We have $D_{out}(v_1) = 5$ and $D_{in}(v_1) = 2$.

emphTVL is controlled by a vertex significance function ρ. Each vertex $v \in V$ is assigned a *unique* significance value $\rho(v)$ to indicate the relative importance of v w.r.t. other vertices. Intuitively, the more time-respecting paths that pass through v, the more important v is. However, computing this vertex significance value requires us to compute the number of time-respecting paths that contain v, which is prohibitively expensive for massive graphs. Instead, we define $\rho(v)$ as the order of v after all the vertices are sorted in descending order of their degrees (i.e., $D_{in}(v) + D_{out}(v)$). As a result, the vertex v with the largest degree has $\rho(v) = 1$, the vertex u with the second largest degree has $\rho(u) = 2$, and so on. When there is a tie, e.g., both v_1 and v_2 have the n-th largest degree, v_1 is assigned $\rho(v_1) = n$ and v_2 is assigned $\rho(v_2) = n + 1$.

Theoretically, the reason why we define vertex significance values $\rho(v)$ by vertex degrees is that the larger the degree v has, the higher the possibility is that v can reach or be reached by other vertices, and the more important v is. We assign the value of $\rho(v)$ in descending order of vertex degrees. This assignment ensures that important vertices are maintained by *TVL* so that many reachable vertex pairs can be answered by *TVL*. We shall see in Section 6 that defining $\rho(v)$ according to vertex degrees results in more efficient querying, compared with assigning $\rho(v)$ randomly.

Based on the concept of vertex significance values $\rho(v)$, we define the temporal in-labels and temporal out-labels of vertex v below.

Definition 7 Given a temporal graph $G = (V, E)$, a vertex v, a vertex significance function ρ, and a time-respecting path p from another vertex u ($\neq v$) to v, a ***temporal in-label*** l_{in} ***of vertex*** $\boldsymbol{v}$ is defined w.r.t. p in the form of $(minV, sig, s_t, a_t)$, where $minV$ is a vertex on p, $sig = \rho(minV)$, s_t is the start time from $minV$, and a_t is the end time at v. Such an in-label implies that there is a vertex $minV$ passed by p (i.e., $minV \in p$) such that (i) $minV \neq v$ and (ii) $\nexists v' \in p$ having $\rho(v') < si$.

Definition 8 Given a temporal graph $G = (V, E)$, a vertex v, a vertex significance function ρ, and a time-respecting path p' from vertex v to another vertex u ($\neq v$), a ***temporal out-label*** l_{out} ***of vertex*** $\boldsymbol{v}$ is defined w.r.t. p' in the form of $(minV', sig', s'_t, a'_t)$, where $minV'$ is a vertex on p', $sig' = \rho(minV')$, s'_t is the start time from vertex v, and a'_t is the end time at vertex $minV'$. Such an out-label implies that there is a vertex $minV'$ passed by p' (i.e., $minV' \in p'$) such that (i) $minV' \neq v$ and (ii) $\nexists v' \in p'$ having $\rho(v') < sig'$.

Note that, each temporal label of vertex v indicates the existence of a vertex ($\neq v$) with the smallest significance value in a time-respecting path to or from v.

Example 5 Consider vertex v_2 in Fig. 2(a). The vertex significance value $\rho(v)$ of vertices v in Fig. 2(a) are given in the column entitled $\rho(v_i)$ of Fig. 4(e). According to the time-respecting path $p = \langle v_1, e_9, v_2 \rangle$, v_2 has an in-label $l_{in} = (v_1, 1, 5, 6)$. According to another time-respecting path $p' = \langle\ v_4,\ e_8,\ v_1,\ e_{10},\ v_2 \rangle$, v_2 has another in-label $l_{in} = (v_1, 1, 5, 7)$. Also notice that not all time-respecting paths generate temporal in-labels or out-labels. For instance, the time-respecting path $p'' = \langle\ v_5,\ e_3,\ v_2 \rangle$ from v_5 to v_2 generates no temporal in-label for v_2, but it does generate a temporal out-label for v_5.

Based on the temporal in-labels and temporal out-labels of vertex v, we define the notions of canonical temporal in-label and out-label as follows.

Definition 9 Given a temporal in-label (resp. out-label) la of vertex v, if there is no other temporal in-label (resp. out-label) la' of v such that $la'.minV = la.minV$ and $[la'.s_t, la'.a_t] \subseteq [la.s_t, la.a_t]$ (i.e., $(la'.s_t \geq la.s_t) \wedge (la'.a_t \leq la.a_t)$), la is defined as a ***canonical temporal in-label*** (resp. ***canonical temporal out-label***).

For example, in the temporal graph in Fig. 2(a), according to Definition 7, v_2 has two temporal in-labels, i.e., $l_{in1} = (v_1, 1, 5, 6)$ and $l_{in2} = (v_1, 1, 5, 7)$. Here, l_{in1}, not l_{in2}, is a canonical temporal in-label of v_2 by Definition 9 because $l_{in1}.s_t = l_{in2}.s_t = 5$ and $l_{in1}.a_t\ (= 6) < l_{in2}.a_t\ (= 7)$. For each vertex v, maintaining all canonical temporal in-labels (resp. out-labels) of v is sufficient to be able to correctly answer temporal reachability queries. Temporal labels like l_{in2} are redundant, and hence are excluded from *TVL*. Even with this reduction, an index is still too large to build when a graph is large. Consequently, we propose a lightweight index *TVL* with a parameter k that controls its size.

Definition 10 (**TVL** Index). Given a temporal graph $G = (V, E)$ and an integer k, a *TVL* index is a labeling scheme, where each vertex $v \in V$ is associated with two sets, $Lin(v)$ and $Lout(v)$. Canonical temporal in-labels (resp. out-labels) la are maintained by $Lin(v)$ (resp. $Lout(v)$) if $la.sig$ is one of the top-k smallest significance values among those of canonical temporal in-labels (resp. out-labels) of v.

Example 6 Given $k = 2$, the *TVL* index of the temporal graph depicted in Fig. 2(a) is shown in the two columns entitled $Lin(v_i)$ and $Lout(v_i)$ in Fig. 4(e).

Algorithm 2: *TVL* Construction Algorithm (LCA)

Input: a temporal graph $G = (V, E)$, an integer k, a vertex significance function ρ
Output: $Lin(v)$ and $Lout(v)$ for every vertex $v \in V$

1: **foreach** vertex $v \in V$ **do**
2: **if** superstep $= 0$ **then**
3: **foreach** $e_{out} = (v, w, s_t, a_t)$ with $\rho(v) < \rho(w)$ **do**
4: $tag \leftarrow 0$; send $(v, \rho(v), s_t, a_t, tag)$ to w
5: **foreach** $e_{in} = (u, v, s_t, a_t)$ with $\rho(v) < \rho(u)$ **do**
6: $tag \leftarrow 1$; send $(v, \rho(v), s_t, a_t, tag)$ to u
7: **else if** there is no messages in transmit **then**
8: **return** $Lin(v)$ *and* $Lout(v)$
9: **else**
10: **foreach** $msg = (minV, sig, s_t, a_t, tag)$ **do**
11: $flag$ = true
12: **if** $msg.tag = 0$ **then** // compute $Lin(v)$
13: **if** the number of unique $minV$ in $Lin(v) < k$ or $msg.sig \leq$ the largest significance value in $Lin(v)$ **then**
14: $flag \leftarrow$ updateLabel$(msg, Lin(v), k)$
15: **if** $flag$ = true **then** // send message
16: **foreach** edge $e_{out} = (v, w, s_t, a_t)$ with $msg.sig < \rho(w)$ **do**
17: **if** $msg.a_t \leq e_{out}.s_t \wedge H_o.size < k$ **then**
18: $msg.a_t \leftarrow e_{out}.a_t$; send msg to w
19: **else if** $msg.a_t \leq e_{out}.s_t$ and $msg.sig \leq H_o.peek()$ **then**
20: $msg.a_t \leftarrow e_{out}.a_t$; send msg to w
21: update H_o with $msg.sig$
22: **else if** $msg.tag = 1$ **then**//compute $Lout(v)$
23: **if** the number of unique $minV$ in $Lout(v) < k$ or $msg.sig \leq$ the largest significance value in $Lout(v)$ **then**
24: $flag \leftarrow$ updateLabel$(msg, Lout(v), k)$
25: **if** $flag$ = true **then** // send message
26: **foreach** $e_{in} = (u, v, s_t, a_t)$ with $msg.sig < \rho(u)$ **do**
27: **if** $msg.s_t \geq e_{in}.a_t \wedge H_i.size < k$ **then**
28: $msg.s_t \leftarrow e_{in}.s_t$; send msg to u
29: **else if** $msg.s_t \geq e_{in}.a_t$ and $msg.sig \leq H_i.peek()$ **then**
30: $msg.s_t \leftarrow e_{in}.s_t$; send msg to u
31: update H_i with $msg.sig$

Algorithm 3: Function updateLabel (msg, $C(v), k$)

Input: a message msg:($minV$,sig,s_t,a_t,tag), an in-label or out-label set $C(v)$ of v, an integer k
Output: a Boolean *flag* indicating whether msg needs to be sent

1: $flag \leftarrow$ true; $S \leftarrow \{la \in C(v) \mid la.minV = msg.minV\}$ //S is sorted in ascending order of the start time $la.s_t$
2: **if** $S \neq \emptyset$ **then**
3: **foreach** label $la \in S$ **do**
4: **if** $msg.s_t < la.s_t$ and $msg.a_t < la.a_t$ **then**
5: break;
6: **else if** $msg.s_t \leq la.s_t$ and $msg.a_t \geq la.a_t$ **then**
7: $flag \leftarrow$ false; break;
8: **else if** $msg.s_t = la.s_t$ and $msg.a_t < la.a_t$ **then**
9: remove la from $C(v)$; break;
10: **else if** $msg.s_t > la.s_t$ and $msg.a_t \leq la.a_t$ **then**
11: remove la from $C(v)$;
12: **else if** $msg.s_t > la.s_t$ and $msg.a_t > la.a_t$ **then**
13: continue;
14: **if** $flag$ = true **then**
15: insert $(msg.minV, msg.sig, msg.s_t, msg.a_t)$ into $C(v)$
16: **else**
17: insert ($msg.minV$,$msg.sig$,$msg.s_t$,$msg.a_t$) into $C(v)$
18: remove label $la \in C(v)$ with $la.sig >$ k-th largest significance value
19: **return** $flag$

For instance, v_5 has three canonical temporal out-labels: $(v_2, 2, 1, 8)$, $(v_2, 2, 7, 9)$, and $(v_3, 3, 7, 8)$, which are maintained by *TVL* when capturing the top-2 smallest significance values. Also note that the corresponding $Lin(v)$ (resp. $Lout(v)$) sets for some vertices may contain fewer than k canonical temporal in-labels (resp. out-labels), e.g., $Lin(v_2) = \{(v_1, 1, 5, 6)\}$, $Lin(v_3) = \{(v_1, 1, 3, 4)\}$ and $Lout(v_1) = \emptyset$.

5.2 *TVL* construction

To construct the above-defined *TVL* index, we present LCA algorithm. The pseudo-code is shown in Algorithm 2. LCA takes as inputs a temporal graph $G = (V, E)$, an integer k, and a vertex significance function ρ, and it outputs $Lin(v)$ and $Lout(v)$ for every vertex $v \in V$.

First, we consider superstep $= 0$, where all vertices take the following actions. For each temporal edge $e_{out} = (v, w, s_t, a_t)$ from v or $e_{in} = (u, v, s_t, a_t)$ to v, LCA sends a message $msg = (v, \rho(v), s_t, a_t, tag)$ to v's out-neighbor w with $\rho(v) < \rho(w)$ or in-neighbor u with $\rho(v) < \rho(u)$ (lines 3–6). Here, tag is used to differentiate messages. It has two possible values, with value 0 indicating that msg is used for computing $Lin(v)$ and value 1 meaning that msg is employed to compute $Lout(v)$. The execution of the following supersteps proceeds until there is no message to transmit.

The arrival of a message msg with $tag = 0$ at vertex v triggers the re-examination of $Lin(v)$ (lines 9–21). The update is triggered if $Lin(v)$ is not full or msg contains a vertex $minV$ with significance value no larger than that of at least one label maintained by $Lin(v)$. Function updateLabel, with its pseudo-code shown in Algorithm 3, performs the update action, and

returns a Boolean where value *true* indicates that msg represents a potential canonical temporal in-label of v. Consequently, msg needs to be propagated to v's out-neighbors w. To reduce the communication cost, we only send messages having sig value smaller than $\rho(w)$ as no other messages can create new temporal in-labels of w, according to Definition 7.

Similarly, the arrival of a message msg with $tag = 1$ at v triggers the re-examination of $Lout(v)$ (lines 22–31). The update is triggered if $Lout(v)$ is not full or msg contains a vertex $minV$ with significance value no larger than at least that of one label in $Lout(v)$. updateLabel performs the update action, and returns a Boolean, where value *true* means that msg represents a potential canonical temporal out-label of v. Therefore, msg needs to be propagated to v's in-neighbors u having $\rho(u) > msg.sig$. To further reduce the number of messages, H_o (resp. H_i), a priority queue containing at most k $msg.sig$ sent through e_{out} (resp. e_{in}) in descending order, is maintained by e_{out} (resp. e_{in}).

Function updateLabel, i.e., Algorithm 3, takes as inputs a message $msg = (minV, sig, s_t, a_t, tag)$, an in-label or out-label set $C(v)$ of vertex v, and an integer k, and it outputs a Boolean indicating whether or not msg needs to be transmitted.

First, updateLabel initializes the Boolean variable $flag$ to true and lets S be a label set to be used for maintaining temporal labels la having $la.minV = msg.minV$. Note that S is sorted in ascending order of the start time $la.s_t$ (line 1).

Next, if S is not empty, then updateLabel updates the content of $C(v)$ to ensure that $C(v)$ only maintains the current canonical temporal labels with top-k significance values (lines 2–15). Specifically, there are 5 cases, which are covered below.

Case (i): as illustrated in Fig. 3(a), if $msg.s_t < la.s_t$ and $msg.a_t < la.a_t$ (line 4), then $[msg.s_t, msg.a_t]$ is not a subset or a superset of any intervals $[la.s_t, la.a_t]$ of temporal labels la in S, because S is sorted in ascending order of the start time $la.s_t$. Therefore, the current ($msg.minV$, $msg.sig$, $msg.s_t$, $msg.a_t$) is a canonical temporal label according to Definition 9. Then updateLabel breaks from the for-loop (line 5), jumps to line 14, and inserts ($msg.minV$, $msg.sig$, $msg.s_t$, $msg.a_t$) into the set $C(v)$ (line 15).

Case (ii): as depicted in Fig. 3(b), if $msg.s_t \leq la.s_t$ and $msg.a_t \geq la.a_t$ (line 6), this means that $[msg.s_t, msg.a_t] \supseteq [la.s_t, la.a_t]$. Therefore, ($msg.minV, msg.sig$, $msg.s_t, msg.a_t$) cannot be a canonical temporal in-label or out-label of v according to Definition 9. Thus, msg is discarded, and updateLabel sets $flag$ to false (line 7).

Case (iii): as shown in Fig. 3(c), if $msg.s_t = la.s_t$ and $msg.a_t < la.a_t$ (line 8), then $[la.s_t, la.a_t] \supset [msg.s_t, msg.a_t]$, and any interval of the remaining temporal labels in S is not a superset of $[msg.s_t, msg.a_t]$. This indicates that, instead of la, the current ($msg.minV$, $msg.sig$, $msg.s_t$, $msg.a_t$) is a canonical temporal label. Therefore, updateLabel removes la from $C(v)$ (line 9) and then inserts ($msg.minV$, $msg.sig$, $msg.s_t$, $msg.a_t$) into $C(v)$ (line 15).

Case (iv): as plotted in Fig. 3(d), if $msg.s_t > la.s_t$ and $msg.a_t \leq la.a_t$ (line 10), then $[la.s_t, la.a_t] \supset [msg.s_t, msg.a_t]$, and there may be other temporal labels (e.g., la' shown in Fig. 3(d)) whose intervals are supersets of $[msg.s_t, msg.a_t]$. Hence, la is removed from $C(v)$ (line 11), and the for-loop proceeds until all temporal labels whose intervals are supersets of $[msg.s_t, msg.a_t]$ are removed from $C(v)$.

Case (v): as illustrated in Fig. 3(e), if $msg.s_t > la.s_t$ and $msg.a_t > la.a_t$(line 12), then $[la.s_t, la.a_t] \not\supset [msg.s_t, msg.a_t]$. Thus, la should be retained. Since updateLabel is not sure whether or not ($msg.minV$, $msg.sig$, $msg.s_t$, $msg.a_t$) is a canonical temporal label, the for-loop continues (line 13).

Otherwise, S is empty, updateLabel first inserts a temporal label ($msg.minV$, $msg.sig$, $msg.s_t$, $msg.a_t$) into $C(v)$ (line 17), and then removes labels l_a from $C(v)$ if $la.sig$ exceeds the current k-th largest significance value in $C(v)$ (line 18).

Finally, updateLabel returns $flag$ (line 19).

In the following, we illustrate how LCA computes TVL below.

Example 7 Take Fig. 2(a) as an example. Let $k = 2$. First, we consider superstep = 0. For each vertex v_i ($1 \leq i \leq 5$), messages are sent to out-neighbors and in-neighbors whose significance values are larger than $\rho(v_i)$, as illustrated in Fig. 4(a).

At superstep = 1, each v_i updates $Lin(v_i)$ and $Lout(v_i)$ based on the incoming messages, and then sends messages selectively, as depicted in Fig. 4(b). We show how v_3 updates labels and sends messages. Vertex v_3 receives three messages. For $(v_1, 1, 3, 4, 0)$, LCA inserts $(v_1, 1, 3, 4)$ into $Lin(v_3)$ by invoking function updateLabel. It then sends message $(v_1, 1, 3, 9, 0)$ to its out-neighbor v_2 because $\rho(v_1) < \rho(v_2)$ and the end time (= 4) of $(v_1, 1, 3, 4, 0)$ is less than the start time (= 8) of edge e_4. For $(v_1, 1, 4, 6, 1)$ and $(v_2, 2, 8, 9, 1)$, LCA adds $(v_1, 1, 4, 6)$ and $(v_2, 2, 8, 9)$ to $Lout(v_3)$, and sends message $(v_2, 2, 7, 9, 1)$ to its in-neighbor v_5, since $\rho(v_2) < \rho(v_5)$ and the start time (= 8) of $(v_2, 2, 8, 9, 1)$ is equal to the end time (= 8) of e_2.

At superstep = 2, v_2, v_4, and v_5 receive messages. At v_2, message $(v_1, 1, 3, 9, 0)$ has no effect on $Lin(v_2)$, because an in-label $(v_1, 1, 5, 6)$ exists with $5 > 3$ and $6 < 9$. At v_4, message $(v_3, 3, 2, 8)$ is inserted into $Lout(v_4)$. At v_5, message $(v_2, 2, 7, 9)$ is inserted into $Lout(v_5)$, since

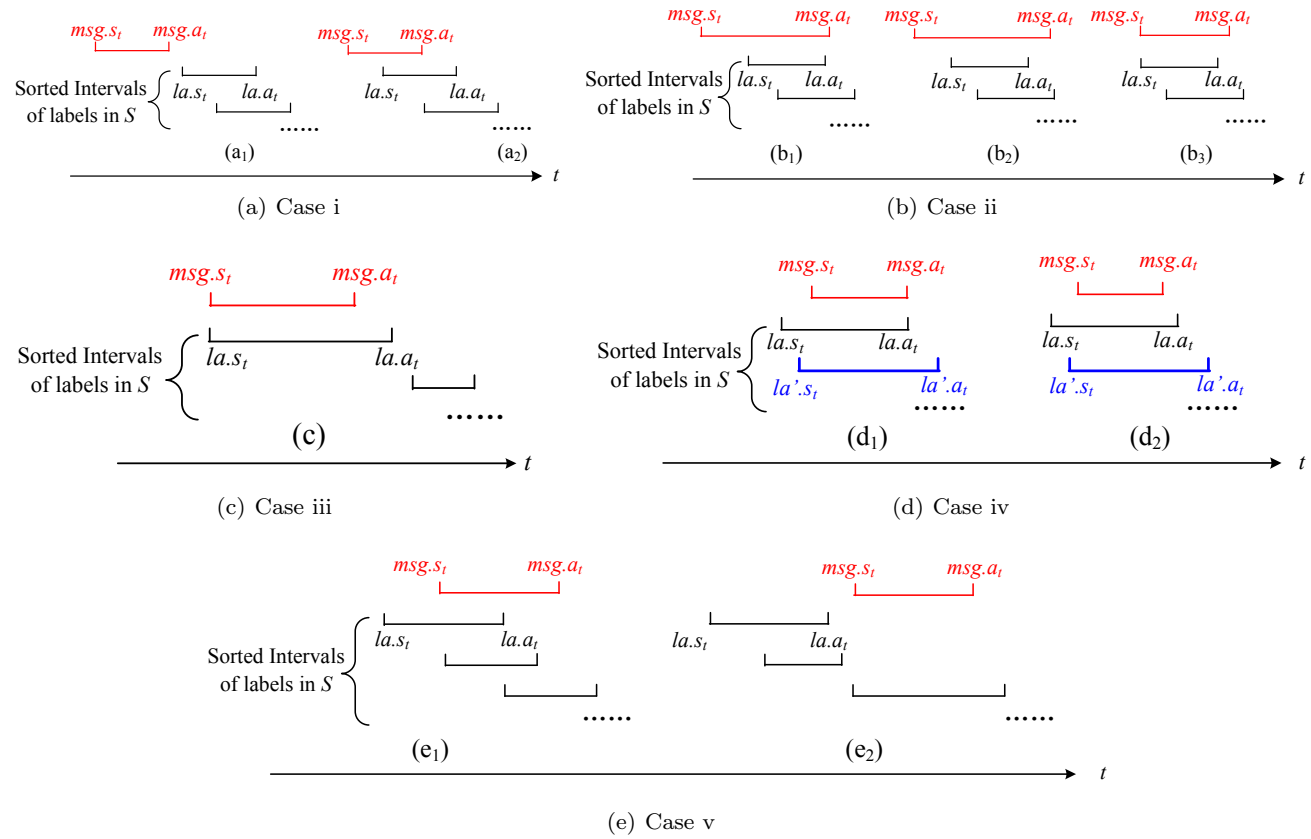


Fig. 3 Illustration of updateLabel

it is a canonical temporal out-label and $\rho(v_2)$ is the second smallest significance value. Moreover, v_5 sends message $(v_2, 2, 2, 9, 1)$ to its in-neighbor v_4, as $\rho(v_2)$ $(= 2)$ is smaller than the second largest significance value $\rho(v_3)$ $(= 3)$ in $Lin(v_4)$ and the start time $(= 7)$ of message $(v_2, 2, 7, 9, 1)$ exceeds the end time $(= 3)$ of edge e_1. The result is shown in Fig. 4(c).

At superstep $= 3$, v_4 receives a message $(v_2, 2, 2, 9, 1)$, and LCA removes $(v_3, 3, 2, 8)$ from $Lout(v_4)$ and inserts $(v_2, 2, 2, 9)$ into $Lout(v_4)$. The result is depicted in Fig. 4(d). Thereafter, since there are no more messages in transmit, LCA stops.

The final TVL index of the temporal graph illustrated in Fig. 2(a) is shown in the two columns entitled $Lin(v_i)$ and $Lout(v_i)$ in Fig. 4(e).

Next, we analyze the correctness and the computation and communication costs of LCA, as well as the size of TVL.

Correctness. We first prove that LCA (i.e., Algorithm 2) correctly computes the temporal in-label set $Lin(v)$ for each vertex $v \in V$. As stated in lines 3–4 of Algorithm 2, for a vertex $v \in V$, a message $(v, \rho(v), s_t, a_t, 0)$ is initialized for each outgoing edge $e_{out} = (v, w, s_t, a_t)$, and then is sent to w, one of v's out-neighbors. Here, if $\rho(v) > \rho(w)$, LCA will not initialize or send the message. This is because $(v, \rho(v), s_t, a_t)$ cannot be contained in $Lin(w)$ and it is useless for computing Lin sets of w's descendants according to Definition 7. Then, as stated in lines 10–21 of Algorithm 2, each vertex v computes $Lin(v)$ iteratively and we need to guarantee that the computation process detailed in lines 10–21 of Algorithm 2 correctly computes $Lin(v)$ in the TVL index. To be more specific, $Lin(v)$ is computed according to the messages sent from (i) v's 1-hop in-neighbors; and (ii) v's multiple-hop in-neighbors.

For the messages sent from v's 1-hop in-neighbors (i.e., case (i)), all the needed messages (i.e., $(u, \rho(u), s_t, a_t, 0)$ with $\rho(u) < \rho(v)$ from 1-hop in-neighbors) are sent according to Definition 7, as stated in lines 3–4 of Algorithm 2. The messages are processed by Algorithm 3. Lines 3-17 of Algorithm 3 ensure that current canonical temporal in-labels are maintained by $Lin(v)$ according to Definition 9. Line 18 of Algorithm 3 guarantees that the current canonical temporal in-labels with top-k smallest significance values are maintained by $Lin(v)$ according to Definition 10.

For the messages propagated from v's multiple-hop in-neighbors (i.e., case (ii)), multiple-hop in-neighbors

vertex	outgoing messages <id, message>
v_1	v_3 (v_1,1,3,4,0) (v_1,1,4,6,1) v_4 (v_1,1,6,7,0) (v_1,1,2,4,0) (v_1,1,4,5,1) v_2 (v_1,1,5,6,0) (v_1,1,5,7,0)
v_2	v_3 (v_2,2,8,9,1) v_5 (v_2,2,1,8,1)
v_3	v_5 (v_3,3,7,8,1)
v_4	v_5 (v_4,4,2,3,0)

(a) Superstep = 0

vertex	$Lin(v)$	$Lout(v)$	incoming messages	outgoing messages
v_1	$\varnothing$	$\varnothing$		
v_2	{(v_1,1,5,6)}	$\varnothing$	(v_1,1,5,6,0) (v_1,1,5,7,0)	
v_3	{(v_1,1,3,4)}	{(v_1,1,4,6), (v_2,2,8,9)}	(v_1,1,3,4,0) (v_1,1,4,6,1) (v_2,2,8,9,1)	v_2 (v_1,1,3,9,0) v_5 (v_2,2,7,9,1)
v_4	{(v_1,1,6,7), (v_1,1,2,4)}	{(v_1,1,4,5)}	(v_1,1,2,4,0) (v_1,1,6,7,0) (v_1,1,4,5,1)	
v_5	{(v_4,4,2,3)}	{(v_2,2,1,8), (v_3,3,7,8)}	(v_4,4,2,3,0) (v_2,2,1,8,1) (v_3,3,7,8,1)	v_4 (v_3,3,2,8,1)

(b) Superstep = 1

vertex	$Lin(v)$	$Lout(v)$	incoming messages	outgoing messages
v_2	{(v_1,1,5,6)}	$\varnothing$	(v_1,1,3,9,0)	
v_4	{(v_1,1,6,7), (v_1,1,2,4)}	{(v_1,1,4,5), (v_3,3,2,8)}	(v_3,3,2,8,1)	
v_5	{(v_4,4,2,3)}	{(v_2,2,1,8), (v_2,2,7,9), (v_3,3,7,8)}	(v_2,2,7,9,1)	v_4 (v_2,2,2,9,1)

(c) Superstep = 2

vertex	$Lin(v)$	$Lout(v)$	incoming messages	outgoing messages
v_4	{(v_1,1,6,7), (v_1,1,2,4)}	{(v_1,1,4,5), (v_2,2,2,9)}	(v_2,2,2,9,1)	

(d) Superstep = 3

v_i	$\rho(v_i)$	$Lin(v_i)$	$Lout(v_i)$
v_1	1	$\varnothing$	$\varnothing$
v_2	2	{(v_1, 1, 5, 6)}	$\varnothing$
v_3	3	{(v_1, 1, 3, 4)}	{(v_1, 1, 4, 6), (v_2, 2, 8, 9)}
v_4	4	{(v_1, 1, 6, 7), (v_1, 1, 2, 4)}	{(v_1, 1, 4, 5), (v_2, 2, 2, 9)}
v_5	5	{(v_4, 4, 2, 3)}	{(v_2, 2, 1, 8), (v_2, 2, 7, 9), (v_3, 3, 7, 8)}

(e) Final TVL index on graph G shown in Fig. 2(a)

Fig. 4 Illustration of LCA

propagate two types of messages to v. To ease the discussion, we assume that the message $msg = (minV, sig, s_t, a_t, 0)$ carried by v_1, v's multiple-hop in-neighbor, is propagated to v via a time-respecting path $\langle v_1, e_1, v_2, \cdots, v_n, e_n, v\rangle$. Here, v_1 first propagates msg to v_2. The first type of message satisfies the property that the largest significance value falls in $Lin(v_2) < msg.sig < \rho(v_2)$ and $|Lin(v_2)| = k$. For this type of message, msg is not processed by Algorithm 3 since it is useless for computing $Lin(v_2)$. However, msg can be useful for updating Lin sets of v_1's 2-hop out-neighbors v_3 with $msg.sig < \rho(v_3)$, v_1's 3-hop out neighbors v_4 with $msg.sig < \rho(v_4)$ and so on. Consequently, LCA propagates msg to v_1's 2-hop out-neighbors, as stated in lines 16–21 of Algorithm 2. The second type of message satisfies the property that the largest significance value falls in $Lin(v_2) \geq msg.sig$ or $|Lin(v_2)| < k$. For this type of message, msg is useful for updating $Lin(v_2)$, and hence it is processed by Algorithm 3, as stated in lines 13–14 of Algorithm 2. If there exists a canonical temporal in-label l_{in} in $Lin(v_2)$ satisfying $l_{in}.minV = msg.minV$ and $[l_{in}.s_t, l_{in}.a_t] \subseteq [msg.s_t, msg.a_t]$, msg is identified as a redundant message because all the subsequent labels created by msg can be created by l_{in}. Therefore, algorithm 3 does not propagate msg. In summary, both the first and second types of useful messages are propagated to v after n-hop propagation, and thus, $Lin(v)$ is updated correctly by Algorithm 3.

Hence, $Lin(v)$ is computed correctly. Similarly, we can prove that Algorithm 2 computes $Lout(v)$ correctly for every $v \in V$. As the proof is similar, it is omitted.

Computation cost. Given a temporal graph $G = (V, E)$, let $\sharp superstep$ be the total number of supersteps, and $|n_i|$ be the number of vertices that are active (i.e., executing computing tasks) at superstep $= i$. Then, the computation cost of LCA is $\sum_{i=0}^{\sharp superstep-1} |n_i|$, which is the inevitable cost incurred due to the computing model of Pregel-like systems. It is worth mentioning that, $\sharp superstep$ is bounded by $[\lceil \frac{d_{p_{max}}}{2} \rceil, d_{p_{max}}]$, where p_{max} is one of the longest time-respecting paths in G, and $d_{p_{max}}$ is the length of p_{max}. The reason is that in the worst case, the source vertex of p_{max} propagates the messages to the target vertex of p_{max} with $d_{p_{max}}$ supersteps, if the significance value of the source vertex is the smallest among those of vertices in p_{max}. In the best case, the middle vertex of p_{max} propagates the messages to the source/target vertex of p_{max} with $\lceil \frac{d_{p_{max}}}{2} \rceil$ supersteps, which occurs if the significance value of the middle vertex is the smallest among those of the vertices in p_{max}.

Communication cost. Let $|n_{p_i}|$ be the number of time-respecting paths (denoted as p_i) whose lengths are all $i + 1$ at superstep $= i$. Then, in the worst case, the communication cost of LCA is $\sum_{i=0}^{\sharp superstep-1} |n_{p_i}|$, if the significance values of the sequenced vertices in p_i are sorted in ascending or descending order.

Size of *TVL*. Given a temporal graph $G = (V, E)$, the size of TVL is $\sum_{v \in V}(|Lout(v)| + |Lin(v)|)$. Let $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ be the transformed DAG of G. According to Definition 10, $|Lin(v)| \leq \sum_{i=1}^{k} |\pi_i|$ and $|Lout(v)| \leq \sum_{i=1}^{k} |\pi_i|$, where $|\pi_i|$ denotes the number of intervals associated with the i-th smallest significance value. Hence, the size of TVL is bounded by $O((\sum_{i=1}^{k} |\pi_i|)|V|)$. In contrast, as found in [41], the index size of *PTopChain* is bounded by $O(k|\mathscr{V}|)$, which exceeds the maximal size of TVL. This is because $|\mathscr{V}|$ denotes the vertex set size of the transformed graph (i.e., the total number of distinct start and arrival time instances of all edges in E), which is much larger than $|V|$ of the original graph.

5.3 Support for insertion

As the topology of a temporal graph G is likely to vary over time (e.g., a telephone network is updated as new conversations are initiated), it is attractive for an indexing technique to be able to support insertion operations. Next, we consider the insertion of a new temporal edge $e = (x, y, s'_t, a'_t)$.

First, the value of $\rho(x)$ (resp. $\rho(y)$) needs to be determined if x (resp. y) is a newly inserted vertex. To achieve an insertion overhead that is as small as possible, x (resp. y) is assigned the largest possible significance value, i.e., $\rho(x)$ (resp. $\rho(y)$) $= |V| + 1$. This guarantees that all vertices have unique significance values. If x (resp. y) is an existing vertex, we do not change its significance value. Although the value may no longer satisfy the definition of significance values, this arrangement is effective at reducing the insertion cost.

Second, let $G' = (V', E')$ be the temporal graph obtained after inserting a new temporal edge $e = (x, y, s'_t, a'_t)$ into $G = (V, E)$. Then, we develop a *TVL Insertion Algorithm* (LIA) to compute or update the label sets $Lin(v)$ and $Lout(v)$ for the affected vertices v. The pseudo-code of LIA is presented in Algorithm 4. LIA takes as inputs an updated temporal graph $G' = (V', E')$, an integer k, a vertex significance function ρ, $Lin(v)$ and $Lout(v)$ for every vertex $v \in V$, and a newly inserted temporal edge $e(x, y, s'_t, a'_t)$, and it outputs updated $Lin(v)$ and $Lout(v)$ for every vertex $v \in V'$.

LIA first computes $R_{in}(y)$ and $R_{out}(x)$, which are potential labels used for updating the labels of affected vertices (lines 1–8). $R_{in}(y)$ contains all vertices u ($\neq y$) with the minimal significance value in a time-respecting path from u to y via edge e, attached with (s_{t_u}, a'_t) where s_{t_u} is the start time from u to y. Similarly, $R_{out}(x)$ includes all vertices w ($\neq x$) with the minimal significance value in a time-respecting path from x to w via edge e, attached with (s'_t, a_{tw}), where a_{tw} is the end time at w. Then, LIA sends $R_{in}(y)$ (resp. $R_{out}(x)$) to y (resp. x). The ensuing process is the same as that detailed in lines 7–31 of Algorithm 2, i.e., computing or updating the label sets $Lin(v)$ and $Lout(v)$ for the affected vertices v. The insertion can be completed with low computational and communication costs, to be studied in the experimental evaluation.

Discussions. The insertion algorithm LIA only supports vertex/edge and new temporal interval insertions on TVL. This is because, in many real-world applications, edge insertions are more frequent than deletions of edges and updates of temporal intervals [21]. When an edge is deleted from the graph or a temporal interval is updated, we need to re-construct TVL index in the worst case. Theoretically, deletion operations are more complicated than insertion operations because for insertion operations, we can easily find the affected vertices. While for deletion operations, it is hard to estimate the number of the affected vertices and tell the affected region, we just have to say in the worst case, we need to re-construct the entire index.

Also notice that, when a new vertex is inserted into $G = (V, E)$, we assign the new vertex the largest possible significance value ($|V| + 1$). This assignment avoids TVL index reconstruction. Since the new vertex has the largest possible significance value, it is not maintained by the Lin or $Lout$ sets of the original vertices, according to Definitions 7 and 8. Hence, the original

Algorithm 4: *TVL* Insertion Algorithm (LIA)

Input: an updated temporal graph $G' = (V', E')$, an integer k, a vertex significance function ρ, $Lin(v)$ and $Lout(v)$ for every vertex $v \in V$, a new temporal edge $e = (x, y, s'_t, a'_t)$
Output: $Lin(v)$ and $Lout(v)$ for every vertex $v \in V'$
1: **if** processing vertex $v = y$ **then**
2: compute $R_{in}(y)$ containing all vertices u ($\neq y$) with the minimal significance value in a time-respecting path from u to y via edge e, attached with (s_{tu}, a'_t). Here, s_{tu} is the start time from u
3: **foreach** vertex u attached with (s_{tu}, a'_t) in $R_{in}(y)$ **do**
4: send $(u, \rho(u), s_{tu}, a'_t, 0)$ to y
5: **if** processing vertex $v = x$ **then**
6: compute $R_{out}(x)$ containing all vertices w ($\neq x$) with the minimal significance value in a time-respecting path from x to w via edge e, attached with (s'_t, a_{tw}). Here, a_{tw} is the end time at w
7: **foreach** vertex w attached with (s'_t, a_{tw}) in $R_{out}(x)$ **do**
8: send $(w, \rho(w), s'_t, a_{tw}, 1)$ to x
9: The remainder is the same as lines 7-31 in Algorithm 2

Lin and *Lout* sets of each vertex remain valid and are treated as the intermediate results during the new *TVL* index update. When inserting a new edge between existing vertices, we do not change the vertex significance values. Although the value may no longer satisfy the definition of significance values, Algorithm 4 still correctly updates the *TVL* index, which follows from the following analysis.

Correctness. After inserting a new edge $e = (x, y, s'_t, a'_t)$ into the temporal graph G, the *Lin* sets of vertices y and y's descendants as well as the *Lout* sets of vertices x and x's ancestors may be updated. Lines 2–4 of Algorithm 4 compute $R_{in}(y)$ according to all time-respecting paths to y via e and then send $R_{in}(y)$ to y. Here, according to Definition 7, $R_{in}(y)$ contains all the messages used for updating *Lin* sets of y and y's descendants. Similarly, lines 6–8 of Algorithm 4 compute $R_{out}(x)$ according to all time-respecting paths from x via e and then send $R_{out}(x)$ to x. Here, according to Definition 8, $R_{out}(x)$ contains all the messages used for updating *Lout* sets of x and x's ancestors. Hence, messages used for updating the *Lin* sets of vertices y and y's descendants as well as messages used for updating the *Lout* sets of vertices x and x's ancestors are sent out. Since the newly inserted vertex with the largest significance value will not be maintained by the *Lin* or *Lout* sets of the original vertices according to Definitions 7 and 8, the original *Lin* and *Lout* sets of each vertex remain valid and are treated as the intermediate results during the new *TVL* index update. Thus, Algorithm 4 creates *Lin* and *Lout* sets for the newly inserted vertices, adds new canonical temporal labels with top-k smallest significance values to original *TVL* index or updates the temporal intervals of the original canonical temporal labels according to Algorithm 2, which has been proven to correctly create a *TVL* index.

In the sequel, we analyze the time and space complexities of LIA.

Time complexity. LIA computes or updates the label sets for the affected vertices. Similar to the complexity analysis of LCA, the computation cost of LIA is $\sum_{i=0}^{\sharp superstep-1} |n_i|$. Here, $|n_i|$ is the number of vertices that are active (i.e., executing computing tasks) at superstep $= i$; $\sharp superstep$ is bounded by $[\lceil \frac{d_{p'_{max}}}{2} \rceil, d_{p'_{max}}]$, where p'_{max} is the longest time-respecting path between all pairs of affected vertices. The communication cost of LIA is $\sum_{i=0}^{\sharp superstep-1} |n_{p_i}|$, where $|n_{p_i}|$ is the number of time-respecting paths (denoted as p_i) whose lengths are all $i+1$ at superstep $= i$. Hence, the time complexity of LIA is $\sum_{i=0}^{\sharp superstep-1}(|n_i| + |n_{p_i}|)$.

Space complexity. In the worst case, the space complexity of LIA is $O(\sum_{v \in V}(|Lin(v)| + |Lout(v)|) + |V| + |E| + \sum_{i=0}^{\sharp superstep-1} |n_{p_i}|)$, where $O(\sum_{v \in V}(|Lin(v)| + |Lout(v)|))$ is the index size, $O(|V| + |E|)$ denotes the size of the temporal graph G and $O(\sum_{i=0}^{\sharp superstep-1} |n_{p_i}|)$ is the message size as analyzed in Section 5.2. In contrast, for *PTopChain*, the space complexity is $O(\sum_{u \in \mathscr{V}}(|Lin(u)| + |Lout(u)|) + |\mathscr{V}| + |\mathscr{E}|)$, where $O(\sum_{u \in \mathscr{V}}(|Lin(u)| + |Lout(u)|))$ denotes the size of *PTopChain* and messages, and $O(|\mathscr{V}| + |\mathscr{E}|)$ is the size of the transformed graph $\mathscr{G}$.

5.4 *TVL* based query algorithms

We now detail how to use *TVL* together with *BFS* to compute the three types of reachability queries defined in Section 3. We first present three lemmas that aim to enable improved efficiency.

Lemma 1 *Given a TVL index built on a temporal graph G and a temporal reachability query $TRQ(u, v, I)$, where $I = [w_s, w_a]$, $TRQ(u, v, I)$ is true if one of the following three conditions holds:* (i) $\exists$ *label* $lu \in Lout(u)$ *having* $lu.minV = v$ *and* $lu.s_t \geq w_s$, $lu.a_t \leq w_a$. (ii) $\exists$ *label* $lv \in Lin(v)$ *having* $lv.minV = u$ *and* $lv.s_t \geq w_s$, $lv.a_t \leq w_a$. (iii) $\exists$ *label* $lu \in Lout(u)$ *and label* $lv \in Lin(v)$ *such that* $lu.s_t \geq w_s$, $lv.a_t \leq w_a$, $lu.minV = lv.minV$, *and* $lu.a_t \leq lv.s_t$.

Proof If condition (i) or condition (ii) holds, $TRQ(u, v, I)$ is obviously true by Definition 3. Thus, we proceed to

prove that $TRQ(u, v, I)$ is also true if condition (iii) holds. A label $lu \in Lout(u)$ guarantees that u can reach $lu.minV$ within $[lu.s_t, lu.a_t]$. A label $lv \in Lin(v)$ ensures that v can be reached by $lv.minV$ within $[lv.s_t, lv.a_t]$. As $lu.minV = lv.minV$ and $lu.a_t \leq lv.s_t$, there is at least one time-respecting path from u to v via $lv.minV$ within $[lu.s_t, lv.a_t]$. Since $lu.s_t \geq w_s$ and $lv.a_t \leq w_a$, $TRQ(u, v, I)$ is true. This completes the proof. □

Example 8 Consider a query $TRQ(v_3, v_4, [1, 7])$ on the temporal graph G shown in Fig. 2(a). There exists $lu = (v_1, 1, 4, 6) \in Lout(v_3)$ and $lv = (v_1, 1, 6, 7) \in Lin(v_4)$ having $lu.minV = lv.minV = v_1$, $lu.s_t\ (= 4) > w_s\ (= 1)$, $lv.a_t = w_a = 7$, and $lu.a_t = lv.s_t = 6$, meaning that condition (iii) of Lemma 1 holds. Therefore, $TRQ(v_3, v_4, [1, 7])$ is true.

Next, to facilitate the presentation of Lemma 2, we introduce four sets $S_o(u)$, $S_i(u)$, $S_o(v)$, and $S_i(v)$. These auxiliary structures are defined corresponding to a reachability query $TRQ(u, v, I)$ with $I = [w_s, w_a]$ on a given temporal graph G. Function $EXTRACT(C, x, con)$ extracts all labels la in a specified label set C such that $la.minV = x$, and Boolean condition con is true.

For each set of labels in $Lout(u)$ having the same $minV$, $\boldsymbol{S_o(u)}$ maintains the label la with the minimal $la.s_t$ among those satisfying $s_t \geq w_s$ (i.e., la is the label having the minimal s_t value among those returned by $EXTRACT(Lout(u), minV, s_t \geq w_s)$).

For each set of labels in $Lin(u)$ having the same $minV$, $\boldsymbol{S_i(u)}$ stores the label la with the maximal $la.a_t$ among those satisfying $a_t \leq s_{t_{min}}$ (i.e., la is the label having the maximal a_t value among those returned by $EXTRACT(Lin(u), minV, a_t \leq s_{t_{min}})$). Here, $s_{t_{min}}$ refers to the minimal start time from u satisfying $s_{t_{min}} \in I$. In particular, if no such $s_{t_{min}}$ exists, $S_i(u) = \emptyset$.

For each set of labels in $Lout(v)$ having the same $minV$, $\boldsymbol{S_o(v)}$ maintains the label la with the minimal $la.s_t$ among those satisfying $s_t \geq a_{t_{max}}$ (i.e., la is the label having the minimal s_t value among those returned by $EXTRACT(Lout(v), minV, s_t \geq a_{t_{max}})$). Here, $a_{t_{max}}$ denotes the maximal end time on v satisfying $a_{t_{max}} \in I$. In particular, if no such $a_{t_{max}}$ exists, $S_o(v) = \emptyset$.

For each set of labels in $Lin(v)$ having the same $minV$, $\boldsymbol{S_i(v)}$ stores the label la with the maximal $la.a_t$ among those satisfying $a_t \leq w_a$ (i.e., la is the label having the maximal a_t value among those returned by $EXTRACT(Lin(v), minV, a_t \leq w_a)$).

Lemma 2 *Given a TVL index built on a temporal graph G and a temporal reachability query $TRQ(u, v, I)$ with $I = [w_s, w_a]$, $TRQ(u, v, I)$ is false if one of the following two conditions holds.* (i) *Let ρ_v and ρ_u be the minimal significance value of any label in $S_o(v)$ and $S_o(u)$, respectively, $\rho_u > \rho_v$ holds.* (ii) *Let ρ'_v and ρ'_u be the minimal significance value of any label in $S_i(v)$ and $S_i(u)$, respectively, $\rho'_u < \rho'_v$ holds.*

Proof First, we prove by contradiction that our statement is true if condition (i) holds. Assume that condition (i) holds, then $TRQ(u, v, I)$ is true, i.e., u can reach v within $I = [w_s, w_a]$. Let $lv \in S_o(v)$ be the label with the minimal significance value (i.e., $lv.sig = \rho_v$). Then v can reach $lv.minV$ within $[lv.s_t, lv.a_t]$, where $lv.s_t \geq a_{t_{max}}$ ($a_{t_{max}}$ is the maximal end time on v satisfying $a_{t_{max}} \in I$). Hence, u can reach $lv.minV$ within $[w_s, lv.a_t]$. Let $lu \in S_o(u)$ be the label with the minimal significance value (i.e., $lu.sig = \rho_u$). Then, as $\rho_u > \rho_v$, $lv \in S_o(u)$ by Definition 10. Consequently, the minimal significance value among those of labels in $S_o(u)$ is ρ_v and not ρ_u, which is a contradiction. Therefore, if condition (i) holds, $TRQ(u, v, I)$ must be false.

Second, we again prove by contradiction that if condition (ii) holds, $TRQ(u, v, I)$ is false. Assume that if condition (ii) holds, then $TRQ(u, v, I)$ is true, i.e., u can reach v within $I = [w_s, w_a]$. Let $lu' \in S_i(u)$ be the label with the minimal significance value (i.e., $lu'.sig = \rho'_u$). Then $lu'.minV$ can reach u within $[lu'.s_t, lu'.a_t]$, where $lu'.a_t \leq s_{t_{min}}$ ($s_{t_{min}}$ is the minimal start time from u satisfying $s_{t_{min}} \in I$). Hence, we have that $lu'.minV$ can reach v within $[lu'.s_t, w_a]$. Let $lv' \in S_i(v)$ be the label with the minimal significance value (i.e., $lv'.sig = \rho'_v$). Then $lu' \in S_i(v)$ by Definition 10 due to $\rho'_u < \rho'_v$. Thus, the minimal significance value among those labels in $S_i(v)$ is ρ'_u and not ρ'_v, which is a contradiction. This completes the proof. □

Example 9 Consider the query $TRQ(v_5, v_3, [1, 4])$ on the graph shown in Fig. 2(a). Since $Lout(v_5) = \{(v_2, 2, 1, 8), (v_2, 2, 7, 9), (v_3, 3, 7, 8)\}$, $EXTRACT(Lout(v_5), v_2, s_t \geq 1) = \{(v_2, 2, 1, 8), (v_2, 2, 7, 9)\}$, and $EXTRACT(Lout(v_5), v_3, s_t \geq 1) = \{(v_3, 3, 7, 8)\}$. It can be seen that $(v_2, 2, 1, 8)$ and $(v_3, 3, 7, 8)$ have the minimal s_t value among those returned by $EXTRACT(Lout(v_5), v_2, s_t \geq 1)$ and $EXTRACT(Lout(v_5), v_3, s_t \geq 1)$, respectively. Then, $S_o(v_5) = \{(v_2, 2, 1, 8), (v_3, 3, 7, 8)\}$. Therefore, the minimal significance value w.r.t. $S_o(v_5)$ is 2 (i.e., $\rho_u = 2$).

Similarly, $S_o(v_3) = \{(v_1, 1, 4, 6), (v_2, 2, 8, 9)\}$, as $(v_1, 1, 4, 6)$ and $(v_2, 2, 8, 9)$ have the minimal s_t value among those returned by $EXTRACT(Lout(v_3), v_1, s_t \geq 4)$ and $EXTRACT(Lout(v_3), v_2, s_t \geq 4)$, respectively. Consequently, the minimal significance value w.r.t. $S_o(v_3)$ is 1 (i.e., $\rho_v = 1$). We have $\rho_u\ (= 2) > \rho_v\ (= 1)$, indicating that condition (i) of Lemma 2 holds. Hence, $TRQ(v_5, v_3, [1, 4])$ is false.

Before detailing Lemma 3, we define an operator $\succ$.

Algorithm 5: *TRQ* Algorithm (TRQA)

```
Input: a temporal graph G = (V, E),
       TVL:{(Lout(v), Lin(v)): ∀v ∈ V}, TRQ(s, t, I)
       with I = [w_s, w_a]
Output: a Boolean indicating whether s can reach t
        within I
1:  foreach vertex v ∈ V do
2:      if superstep = 0 then
3:          if the maximal start time from s < w_s or the
              minimal end time on t > w_a then
4:              return false
5:          else if v = t then
6:              aggregate Lin(t); send w_s to s;
7:          v.minT ← w_a;
8:      else if there is no message in transmit then
9:          return false
10:     else
11:         select minimal w'_s (< v.minT) from messages
12:         if such w'_s exists then
13:             v.minT ← w'_s
14:             if v = t then
15:                 return true
16:             if ∃ label la ∈ Lout(v) having
                  la.minV = t and la.s_t ≥ w'_s and
                  la.a_t ≤ w_a then          //Lemma 1
17:                 return true
18:             if ∃ label la' ∈ Lin(t) having la.minV = t
                  and la'.s_t ≥ w'_s and la'.a_t ≤ w_a then
                  //Lemma 1
19:                 return true
20:             if ∃ labels la ∈ Lout(v) and la' ∈ Lin(t)
                  having la.minV = la'.minV, la.s_t ≥ w'_s,
                  la'.a_t ≤ w_a and la.a_t ≤ la'.s_t then
                  //Lemma 1
21:                 return true
22:             compute S_i(v), S_o(v), S_i(t), and S_o(t)
23:             if the cases of Lemma 2 do not hold then
24:                 if S_o(v) ≻ S_o(t) then    // Lemma 3
25:                     find all labels lv'∈S_o(v) in
                          Lemma 3 and send lv'.a_t to
                          lv'.minV
26:                 else                              //BFS
27:                     foreach edge e_out ← (v, w, s_t, a_t)
                              with [s_t, a_t] ⊆ [w'_s, w_a] do
28:                         send a_t to w
```

Given a reachability query $TRQ(u, v, I)$ with $I = [w_s, w_a]$ and its corresponding label sets $S_o(u)$ and $S_o(v)$, **we say $S_o(u) \succ S_o(v)$** if one of the following two conditions holds: (i) $\exists lu \in S_o(u)$ having $lu.minV = v$ and $lu.a_t > w_a$; (ii) $\exists lu \in S_o(u)$ and $lv \in S_o(v)$ such that $lu.sig = lv.sig$ and $lu.a_t > lv.a_t$.

Lemma 3 *Given a TVL index and a temporal reachability query $TRQ(u, v, I)$ with $I = [w_s, w_a]$, if $S_o(u) \succ S_o(v)$, $TRQ(u, v, I)$ can be answered by $TRQ(lu'.minV, v, [lu'.a_t, w_a])$, where $lu' \in S_o(u)$ having $lu'.sig < lu.sig$. Here, lu is a label that makes operator $\succ$ valid.*

Proof If $S_o(u) \succ S_o(v)$, then at least one of the two conditions defined together with operator $\succ$ needs to hold. If condition (i) holds, u cannot reach v within I via vertex w having $\rho(w) \geq lu.sig$ $(= \rho(v))$. This is because, if u can reach v within I via w having $\rho(w) \geq lu.sig$, it must be that $lu.a_t \leq w_a$ by Definition 10, which contradicts condition (i). Therefore, u can only reach v within I via w having $\rho(w) < lu.sig$, and such vertices w are all maintained by $S_o(u)$ according to Definition 10. Thus, if $S_o(u) \succ S_o(v)$, $TRQ(u, v, I)$ can be answered by $TRQ(lu'.minV, v, [lu'.a_t, w_a])$, where $lu' \in S_o(u)$ having $lu'.sig < lu.sig$. Similarly, we can prove that the above statement is true if condition (ii) holds. This completes the proof. □

Example 10 Consider the query $TRQ(v_5, v_3, [6, 7])$ on the graph G depicted in Fig. 2(a). We have $(lu = (v_3, 3, 7, 8)) \in (S_o(v_5) = \{(v_2, 2, 7, 9), (v_3, 3, 7, 8)\})$ with $lu.minV = v_3$ and $lu.a_t\ (= 8) > w_a\ (= 7)$, i.e., condition (i) of $S_o(v_5) \succ S_o(v_3)$ holds. According to Lemma 3, we are certain that $TRQ(v_5, v_3, [6, 7])$ can be answered by query $TRQ(v_2, v_3, [9, 7])$, which returns false. Next, consider another query $TRQ(v_3, v_4, [3, 4])$. We have $(lu = (v_1, 1, 4, 6)) \in (S_o(v_3) = \{(v_1, 1, 4, 6), (v_2, 2, 8, 9)\})$, and $(lv = (v_1, 1, 4, 5)) \in (S_o(v_4) = \{(v_1, 1, 4, 5)\})$ such that $lu.sig = lv.sig = 1$ and $lu.a_t\ (= 6) > lv.a_t\ (= 5)$, i.e., condition (ii) of $S_o(v_3) \succ S_o(v_4)$ holds. Thus, $TRQ(v_3, v_4, [3, 4])$ can be answered by $TRQ(lu'.minV, v_4, [lu'.a_t, 4])$, where $lu' \in S_o(v_3)$ having $lu'.sig < 1$ by Lemma 3. Since such lu' does not exist, $TRQ(v_3, v_4, [3, 4])$ returns false.

Using the above three lemmas, we present TRQA for supporting temporal reachability queries using *TVL*. The pseudo-code is shown in Algorithm 5. TRQA takes as inputs a temporal graph $G = (V, E)$, the *TVL* index of G (i.e., $\{(Lout(v), Lin(v)): \forall v \in V\}$), and a temporal reachability query $TRQ(s, t, I)$ with $I = [w_s, w_a]$, and it outputs a Boolean indicating whether s can reach t within I.

TRQA is implemented in the distributed graph processing platform Blogel, which distributes vertices to multiple workers that process vertices in parallel. TRQA proceeds in iterations, i.e., supersteps. In each superstep, the vertices at different workers execute the same user-defined function *compute*() (i.e., Lines 2-28 of Algorithm 5) in parallel. The *compute*() function performs the user-specified tasks for each vertex v, including processing v's incoming messages sent from the previous superstep, sending messages to other vertices for the next superstep's computation, and making v vote to

halt. More specifically, first, we consider superstep = 0. If the maximal start time from s is less than w_s or the minimal end time on t is larger than w_a, TRQA returns false since there can be no time-respecting paths between s and t within I (lines 3–4). Otherwise, TRQA aggregates $Lin(t)$ so that $Lin(t)$ becomes available to all vertices in the following supersteps, and it sends w_s to vertex s (lines 5–6). Then, for each vertex v, TRQA initializes a parameter $v.minT$ to w_a (line 7). Note that $v.minT$ indicates the minimal end time sent to v, which can be employed to avoid unnecessary search.

The execution of subsequent supersteps depends on whether there are messages to transmit. If there is no message to transmit, this means that s cannot reach t within I. Thus, TRQA immediately returns false (lines 8–9). Otherwise, a new superstep is performed (lines 10–28). During the new superstep, each vertex v selects the minimal w'_s having $w'_s < v.minT$ from messages for processing (line 11). The reason is that, for the messages $w'_s \geq v.minT$, $TRQ(v, t, [w'_s, w_a])$ can be answered by $TRQ(v, t, [v.minT, w_a])$, which has been issued. If such a w'_s exists, TRQA first updates $v.minT$ (lines 12–13). Then, if the traversed current vertex is t or Lemma 1 applies, TRQA returns true (lines 14–21); otherwise, it computes $S_i(v)$, $S_o(v)$, $S_i(t)$, and $S_o(t)$ (line 22), and utilizes Lemma 2 to examine whether $TRQ(v, t, [w'_s, w_a])$ is false (line 23). If not, Lemma 3 is employed for pruning search space (lines 24–25). If Lemma 3 does not apply, TRQA processes the query by checking if any of the descendants of v can reach t within the corresponding time interval (lines 27–28).

Next, we analyze the correctness of TRQA.

Correctness. TRQA first uses Lemmas 1 and 2 to determine the value of $TRQ(s, t, I)$, which have proven to be correct. Then, in order to find a time-respecting path that connects s and t within I, TRQA searches the whole graph following the Breadth-First-Search paradigm, where Lemma 3 is used to prune the search space. According to Definitions 2 and 3, if a time-respecting path p from s to t satisfying $[S_p, E_p] \subseteq I$ exists, $TRQ(s, t, I)$ is true and TRQA returns true; otherwise, $TRQ(s, t, I)$ is false and TRQA returns false. Let's assume there is a time-respecting path p from s to t satisfying $[S_p, E_p] \subseteq I$ but TRQA returns false. As we employ Lemma 3 to prune paths and Lemma 3 is proved to only prune unqualified paths, the time-respecting path p from s to t satisfying $[S_p, E_p] \subseteq I$ will not be pruned. Thus, TRQA will only return true, which contradicts our assumption. Consequently, our assumption is invalid. Hence, TRQA answers the temporal reachability query correctly.

In addition, when inserting new edges, we do not change the significance values of existing vertices. Although the values may be different from their "true" significance values, the update of TVL will not affect the correctness of TRQA to answer the query $TRQ(s, t, I)$.

In order to prove the correctness of the above statement, we need to explain why we introduce significance values and the role played by significance values during index construction and query processing. 2-hop labeling maintains two entire sets $Lin(v)$ and $Lout(v)$ for each vertex $v \in V$. By $Lin(v)$ and $Lout(v)$, we can immediately find the vertices that can reach v or be reached by v without visiting the whole graph. However, when the graph is big, it might not be possible to maintain the entire $Lin(v)$ and $Lout(v)$ for every vertex v in memory. Hence, significance values are introduced to measure the importance of vertices, and TVL strategically uses $Lin(v)$ and $Lout(v)$ to store only vertices with the top-k smallest significance values, in order to reduce the storage overhead and to improve the index construction efficiency and scalability.

TVL trades query efficiency for the scalability. In other words, TVL only keeps top-k vertices instead of storing the entire $Lin(v)$ and $Lout(v)$, thus, the search based on TVL might incur additional cost to traverse the whole graph when we need to find the entries of $Lin(v)$ and $Lout(v)$ that are not maintained by TVL. More specifically, based on TVL (i.e., the reduced size of $Lin(v)$ and $Lout(v)$), TRQA uses Lemmas 1 and 2 to directly determine the answer of $TRQ(s, t, I)$. However, for cases where $TRQ(s, t, I)$ cannot be determined by Lemmas 1 and 2, TRQA traverses the whole graph following the Breadth-First-Search paradigm, where Lemma 3 is used to prune the search space. The correctness of TRQA (i.e., the correctness of Lemmas 1 to 3) after updating TVL does not depend on the significance values of vertices, but depends on whether the updated $Lin(v)$ and $Lout(v)$ keep correct canonical temporal in-labels and out-labels (i.e., whether the updated index is the correct TVL index). The correctness of the updated index is proven in Section 5.3. Therefore, the vertex significance values will not affect the correctness of TRQA.

We proceed to explain how to answer earliest end time queries and minimum duration queries.

Earliest end time query. An *Earliest End Time Query Algorithm* ($\mathsf{E^2TQA}$) is similar as TRQA. Here, we only point out the differences and skip the detailed pseudo-code of $\mathsf{E^2TQA}$. For TRQA, once $TRQ(s, t, I)$ is true, it returns true, and terminates. In contrast, for $\mathsf{E^2TQA}$, the query needs to examine all time-respecting paths from s to t within I, and then returns the earliest end time. Hence, $\mathsf{E^2TQA}$ initializes a persistent aggregator *ans* that updates the earliest end time once $TRQ(s, t, I)$ is true, i.e., $\mathsf{E^2TQA}$ replaces all statements that return true with an update operation (i.e., updating the earliest end time). Specifically, $\mathsf{E^2TQA}$ replaces

Table 2 Datasets statistics

Graph	$\|V\|$	$\|E\|$	$\|T(G)\|$	$\|\mathscr{V}\|$	$\|\mathscr{E}\|$
Wikipedia	1,870,709	39,953,145	2198	34,814,941	77,196,220
Youtube	3,223,589	9,375,374	203	8,901,388	15,381,556
Flickr	2,302,925	33,140,017	134	12,600,099	44,358,410
Deli-ui	34,611,268	203,597,734	1583	229,502,162	444,017,173
DBLP	46,624,231	463,779,274	207	181,160,598	774,983,634
Syngraph1	131,476,255	800,000,000	1000	1,582,357,528	2,454,205,328
Syngraph2	200,000,000	1,200,000,000	800	2,385,632,635	3,736,045,808

line 9 of Algorithm 5 with a statement that returns ans, replaces line 15 of Algorithm 5 with the statement $ans \leftarrow min(ans, w'_s)$, replaces lines 17 and 19 of Algorithm 5 with the statement $ans \leftarrow min(ans, la.a_t)$, and replaces line 21 of Algorithm 5 with the statement $ans \leftarrow min(ans, la'.a_t)$.

Minimum duration query. Similar to $\mathsf{E^2TQA}$, a *Minimum Duration Query Algorithm* (MDQA) needs to inspect every time-respecting path p from s to t within I. The difference between MDQA and $\mathsf{E^2TQA}$ is that, for every time-respecting path p from s to some w that can reach t within I, $MDQ(s, t, I)$ needs to send (D_p, E_p) to w. An aggregator ans is thus initialized to maintain the minimal duration.

Finally, we analyze the time and space complexities of TRQA, $\mathsf{E^2TQA}$, and MDQA.

Time complexity. In TRQA, $\mathsf{E^2TQA}$, and MDQA, we adopt a simple hash partitioning method to achieve load balancing. Hence, given B_n workers, each worker contains $\frac{|V|}{B_n}$ vertices after hash partitioning. Based on this, the computation time complexity of each superstep is $O(\frac{|V|}{B_n})$, and thus, the computation time complexity of TRQA is $O(\sum_{i=0}^{\sharp superstep-1} \frac{|V|}{B_n})$. Note that, the total number of supersteps $\sharp supersteps$ is not $O(|V| + |E|)$. In the worst case, $\sharp supersteps$ equals the number of edges in a longest time-respecting path that connects vertices s and t. The worst case occurs when all the vertices in the longest time-respecting path are all in different workers. Suppose that N_{B_i} workers need to exchange messages at superstep i. The communication time complexity of TRQA is $O(\sum_{i=0}^{\sharp superstep-1} N_{B_i})$. Hence, the time complexities of TRQA, $\mathsf{E^2TQA}$, and MDQA are $O(\sum_{i=0}^{\sharp supersteps-1} (\frac{|V|}{B_n} + N_{B_i}))$.

Space complexity. The space complexities of TRQA, $\mathsf{E^2TQA}$, and MDQA are $O(\sum_{v\in V}(|Lin(v)|+|Lout(v)|)+|V|+|E|)$, in which $O(\sum_{v\in V}(|Lin(v)|+|Lout(v)|))$ denotes the index size, and $O(|V|+|E|)$ is the size of the temporal graph. The space complexity of *PTopChain* is $O(\sum_{u\in\mathscr{V}}(|Lin(u)|+|Lout(u)|)+|\mathscr{V}|+|\mathscr{E}|)$, where $O(\sum_{u\in\mathscr{V}}(|Lin(u)|+|Lout(u)|))$ denotes the index size, $O(|\mathscr{V}|+|\mathscr{E}|)$ is the size of the transformed graph, and $O(|\mathscr{V}|)$ represents the size of a table that maps vertices in G to those in the transformed graph $\mathscr{G}$. Thus, the space complexities of our query methods are much smaller than that of the baseline.

6 Experimental evaluation

We conduct extensive experiments to evaluate the efficiency and scalability of the proposed methods.

6.1 Experimental settings

Datasets. We use 5 real temporal graphs, in which 4 datasets (i.e., *Wikipedia*, *Youtube*, *Flickr*, *Deli-ui*) are from the Koblenz Large Network Collection[3] and *DBLP* is extracted from the DBLP bibliography. We also use 2 synthetic graphs generated by JTGraph[4]. Statistics for the graphs $G = (V, E)$ and transformed graphs $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ are summarized in Table 2, where $|\mathscr{V}|$ is the number of vertices in $\mathscr{G}$, $|\mathscr{E}|$ is the number of edges in $\mathscr{G}$, and $T(G)$ is the number of temporal intervals in the original temporal graph G. *Wikipedia* is a hyperlink network of the English Wikipedia. It contains users and pages connected by edit events. Each edge represents an edit associated with the edit time. *Youtube* and *Flickr* are social networks, where vertices represent users and edges capture friendships attached with connect time. *Deli-ui* is a bipartite graph containing user-tag relations and user-URL relations extracted from the network "Delicious". *DBLP* is a bibliography network that provides a comprehensive list of research papers. An Edge from a publication A to another publication B represents that A is cited by B, and the edge is annotated with the publication date of B. *Syngraph1* is a random graph created by the Erdős-Rényi model, and *Syngraph2* is a synthetic graph with power-law degree distributions. Edges in *Syngraph1* and *Syngraph2* are randomly assigned time information.

Query sets. For every dataset, we generate 500 queries by selecting a pair of vertices uniformly at random and a set $I = [0, \infty]$ for all queries so that query

[3] KONECT is available at `konect.uni-koblenz.de/`.

[4] JTGraph is available at `http://www.cse.psu.edu/~kxm85/software/GTgraph/`.

processing can access the whole graph. The average query time of each method is reported.

Methods. We compare *TVL* implemented in Blogel with the following approaches:

(i) *PTopChain* (described in Section 4), which is an optimized parallel version of the state-of-the-art approach *TopChain* [41].

(ii) *GTopChain*, which is another parallel version of *TopChain* [41] implemented in GTimer[5].

(iii) Two state-of-the-art distributed methods for general graphs, namely, *DSR* [11] and *disReach* [8].

(iv) An optimized bi-directional breadth-first search (*BFS*) strategy (coined "*Bi-BFS*") that works without an index.

In addition, we also compare *TVL* with *TOL* [52] for evaluating index insertion performance, and compare *TVL* with *TopChain* and the representative centralized scalable method *Grail* for scalability test. All approaches are applied after transforming a temporal graph G into a general graph $\mathscr{G}$, except in the cases of *TVL* and *Bi-BFS*.

General setup. Our distributed algorithms are implemented in Blogel, which is deployed with MPICH 3.2.1. All programs were compiled using GCC 5.4.0 with option -O3 enabled. We conducted the experiments on a 14-node Dell cluster, where one serves as a master and the remaining nodes serve as workers. Cluster nodes are connected via 10GBit LAN. Each node has two 12-core processors, 128GB RAM, and 3TB disk space.

We first investigate index construction and insertion costs and compare index sizes across all methods. Then we consider the performance of three types of reachability queries on top of *TVL*. We note that a graph of certain size calls for a certain amount of computing resources (i.e., workers) to achieve good performance. Therefore, we fix the number of workers to 10 for the middle-scale datasets (i.e., *Wikipedia*, *Youtube*, *Flickr*), and fix the number of workers to 80 for the large-scale datasets (i.e., *Deli-ui*, *DBLP*, and the synthetic graphs). We set $k = 3$ for *TVL* and *PTopChain*. Also, we use bold values in the tables to highlight the best results and "—" to indicate that an index cannot be constructed within 48 hours or a method cannot run.

6.2 Index construction and insertion costs

Table 3 presents the index sizes. Note that *Bi-BFS* does not require an index; thus, we compare *TVL* against *GTopChain*, *PTopChain*, *DSR*, and *disReach*.

[5] GTimer is available at `http://www.cse.cuhk.edu.hk/systems/graph/Gtimer/index.html`.

Table 3 Index size (in GB)

Graph	*TVL*	*PTopChain*	*GTopChain*	*DSR*	*disReach*
Wikipedia	**0.484**	1.127	1.127	—	—
Youtube	**0.142**	0.262	0.262	28.610	8.145
Flickr	**0.229**	0.380	0.380	—	22.626
Deli-ui	**4.180**	7.160	7.160	—	—
DBLP	**3.606**	5.269	5.269	—	—
Syngraph1	**9.230**	49.873	49.873	—	—
Syngraph2	**15.984**	75.527	75.527	—	—

Table 4 Index construction time (in minutes)

Graph	*TVL*	*PTopChain*	*GTopChain*	*DSR*	*disReach*
Wikipedia	**1.252**	7.944	11.850	—	—
Youtube	**0.231**	0.520	0.829	10.754	11.317
Flickr	**0.641**	1.019	5.496	—	26.050
Deli-ui	**9.614**	36.922	37.262	—	—
DBLP	**11.100**	12.585	45.020	—	—
Syngraph1	**20.744**	63.470	61.729	—	—
Syngraph2	**15.295**	29.020	23.854	—	—

The first observation is that, for all temporal graphs, the index sizes of *TVL*, *GTopChain* and *PTopChain* are much smaller than those of *DSR* and *disReach*. The reason is that, both *disReach* and *DSR* need to materialize the pairwise reachability information among the in-boundaries and out-boundaries for each partition, which results in $O(\sum_{j=1}^{k} |I_j||O_j|)$ and $O(\sum_{i=1}^{k}\sum_{j=1,j\neq i}^{k} (|I_i||O_j|+|E_C|))$ space overhead, respectively. Here, $|I_j|$, $|O_j|$, and $|E_C|$ are the cardinality of the in-boundaries, out-boundaries, and cut edges, respectively. In contrast, the index sizes of *PTopChain*, *GTopChain* and *TVL* are linear to the graph size.

The second observation is that the index size of *TVL* is up to 4 times smaller than that of *PTopChain* or *GTopChain*. This is because *PTopChain* and *GTopChain* are constructed based on the transformed graph $\mathscr{G}$; thus, the total label size of *PTopChain* and *GTopChain* are bounded by $O(k|\mathscr{V}|)$. While *TVL* is built based on the original temporal graph G, as analyzed in Section 5.2, the index size of *TVL* is bounded by $O((\sum_{i=1}^{k} |\pi_i|)|V|)$, which is smaller than $O(k|\mathscr{V}|)$.

The third observation is that *DSR* and *disReach* construction algorithms cannot run on most of graphs, which is due to two reasons. First, the transformed graph is a directed acyclic graph, which is difficult to condense using the optimization technique of *DSR*. Second, the boundary graphs are too large to construct even when the transformed graph is partitioned by METIS, not to mention sending them to other partitions for constructing compound graphs.

Next, Table 4 reports the index construction time of *TVL*, *PTopChain*, *GTopChain*, *DSR*, and *disReach*. The first observation is that *TVL* has the smallest index

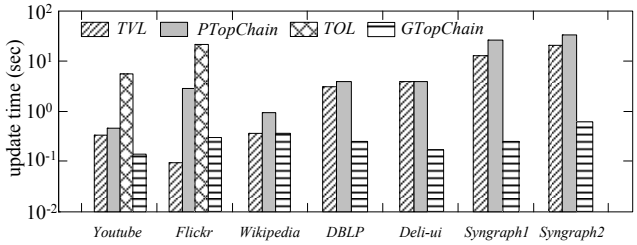


Fig. 5 Average insertion time due to edge insertion

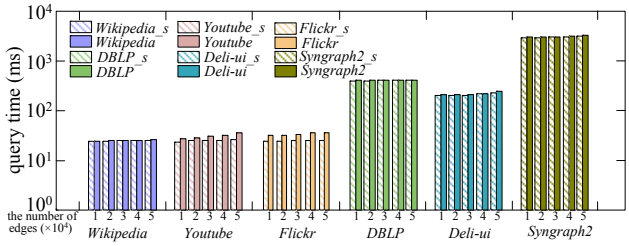


Fig. 6 Query performance vs. the number of inserting edges

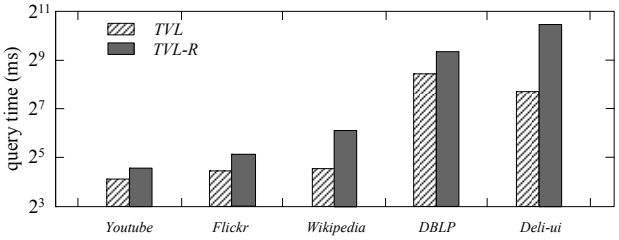


Fig. 7 Query costs vs. different vertex significance functions

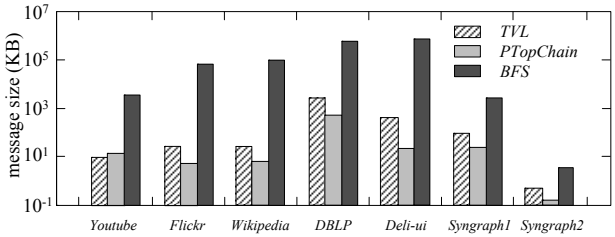


Fig. 8 Average communication cost of *TRQ*

construction overheads on all datasets. This is because *TVL* has smaller index size and is constructed based on the original temporal graph instead of the transformed graph, which is tens of times larger. The second observation is that, *DSR* and *disReach* have much larger index construction costs than *PTopChain*, *GTopChain* and *TVL*. The reason is that *DSR* and *disReach* need to construct sizable boundary graphs.

In addition, we evaluate the average index insertion performance when 500 temporal edges are inserted into the input temporal graph. Fig. 5 shows the average insertion costs of *TVL*, *GTopChain*, *PTopChain*, and *TOL*. The first observation is that the insertion efficiency of *GTopChain* is the best in most cases, as Gtimer is suitable for processing temporal graphs that keep updating over time. The second observation is that the insertions for *TVL* are faster than the insertions for *PTopChain*. For example, on *Flickr*, insertions for *TVL* are one order of magnitude faster than for *PTopChain*. The reason is that inserting a temporal edge inevitably triggers update of the transformed *DAG*, and thus *PTopChain* needs to recompute the topological-sort-based labels, resulting in higher insertion costs.

The third observation is that, on *Youtube* and *Flickr*, the insertion cost for *TOL* is one order of magnitude higher than that for *TVL* and *PTopChain*. Moreover, the indexing time of *TOL* on *Youtube* and *Flickr* is 13069.56 seconds (i.e., 3h 37min 49.56s) and 13,193.20 seconds (i.e., 3h 39min 53.2s), respectively, which is two orders of magnitude longer than that of *TVL* and *PTopChain*. We omit *TOL* on the other 5 datasets because the index of *TOL* cannot be constructed within 2 days. We conclude that *TVL*, *PTopChain* and *GTopChain* are more scalable than *TOL*, and they can achieve much higher insertion efficiency.

We also compare the query performance on the updated index after a bunch of insertions with that on the index built directly from the updated data. Fig. 6 plots the query performance when varying the number of inserted edges. We use $*\_s$ to denote the query performance on the index built directly from the updated data. As can be seen, the query performance based on updated *TVL* degrades only slightly. This is because the significance values of part of the vertices after updates are not real values as defined, which offers evidence of the effectiveness of the significance definition. In addition, the query cost increases with the number of inserted edges because of the growing size of the temporal graph.

The *TVL* index is governed by the vertex significance function ρ, which is defined by vertex degrees. We experimentally assess whether defining $\rho(v)$ by vertex degrees results in more efficient querying compared with assigning $\rho(v)$ randomly. The *TVL* index constructed under randomly assigned vertex significance values is coined *TVL-R*. As depicted in Fig. 7, *TVL* outperforms *TVL-R* on all real datasets.

6.3 Search performance

Evaluating *TRQ*. The first set of experiments studies the query efficiency of *TRQ*. The average query time of each method is shown in Table 5. The percentage of reachability queries that are effectively handled by Lemmas 1 to 3 is reported in Table 6.

Table 5 Query time of *TRQ* (in seconds)

Graph	*TVL*	*PTopChain*	*GTopChain*	*Bi-BFS*	*DSR*	*disReach*
Wikipedia	**0.023**	0.233	0.195	3.034	—	—
Youtube	**0.017**	0.043	0.125	0.370	7.314	912
Flickr	**0.022**	0.080	0.126	1.841	—	7358
Deli-ui	0.211	0.602	**0.191**	19.658	—	—
DBLP	**0.347**	0.546	0.362	13.559	—	—
Syngraph1	0.796	5.431	**0.533**	4.200	—	—
Syngraph2	3.001	9.280	**0.604**	4.915	—	—

Table 6 Percentage of queries handled by Lemmas

Graph	Lemma 1	Lemma 2	Lemma 3
Wikipedia	10.6%	8.4%	2%
Youtube	1.6%	6%	4%
Flickr	50%	6%	21%
Deli-ui	50%	4.4%	1.6%
DBLP	4.16%	10%	4.18%
Syngraph1	0%	13.8%	21.8%
Syngraph2	0%	83.2%	0%

The first observation is that *TVL* achieves the highest efficiency on small-size datasets (including *Wikipedia*, *Youtube*, and *Flickr*), while *GTopChain* achieves the highest efficiency on large-size datasets. This is because, Gtimer is a distributed platform optimized for temporal graphs, while Blogel is a distributed framework targeting traditional graphs. The second observation is that besides *GTopChain*, *TVL* outperforms the other methods on all datasets due to the effective pruning enabled by Lemmas 1 through 3, as depicted in Table 6. In particular, on *Flickr* and *Deli-ui*, 50% of all queries are effectively handled by Lemma 1, i.e., 50% of all query results are directly returned by Lemma 1. On *Syngraph2*, 83.2% of all queries are effectively tackled by Lemma 2, i.e., 83.2% of all query results are directly returned by Lemma 2. On *Flickr* and *Syngraph1*, 21% and 21.8% of all queries are effectively handled by Lemma 3, respectively, i.e., the search space of 21% and 21.8% of all queries is pruned by Lemma 3, respectively. The third observation is that *Bi-BFS* is able to run on all datasets, but is one order of magnitude slower than *TVL* on average. This is because it needs more supersteps to complete the search, resulting in increased communication cost. as can be seen in Fig. 8. *DSR* and *disReach* are several orders of magnitude slower than *TVL*, which limits their scalability. Consequently, they are unable to run on most of the datasets. Note that, although the inter-machine communication cost of *TVL* is much higher than that of *PTopChain* in most cases (as plotted in Fig. 8), *TVL* still achieves better performance than *PTopChain* (as shown in Table 5). The reason is that, the input to *PTopChain* is the transformed *DAG*s of the original graphs, which are tens of times larger. Therefore, every worker is assigned a larger local subgraph, resulting in higher computational costs that exceed the communication costs. These findings indicate that *TVL*'s labeling scheme is an efficient and scalable approach.

Table 7 Positive query time of *TRQ* (in seconds)

Graph	*TVL*	*PTopChain*	*Bi-BFS*
Wikipedia	**0.019**	0.158	0.727
Youtube	**0.040**	0.094	0.464
Flickr	**0.026**	0.084	0.757
Deli-ui	**0.164**	0.416	7.367
DBLP	**0.360**	0.943	12.351
Syngraph1	2.282	20.814	**1.607**
Syngraph2	**2.913**	12.006	3.116

Table 8 Negtive query time of *TRQ* (in seconds)

Graph	*TVL*	*PTopChain*	*Bi-BFS*
Wikipedia	**0.028**	0.309	5.436
Youtube	**0.016**	0.042	0.368
Flickr	**0.020**	0.079	1.978
Deli-ui	**0.260**	0.790	32.554
DBLP	**0.344**	0.520	13.644
Syngraph1	**0.790**	5.400	4.210
Syngraph2	**3.024**	8.794	4.972

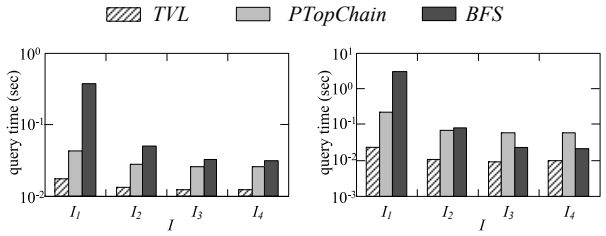
Table 9 Query time of *MDQ* (in seconds)

Graph	*TVL*	*PTopChain*	*GTopChain*	*Bi-BFS*
Wikipedia	**0.145**	0.754	0.268	23.799
Youtube	**0.021**	0.048	0.120	0.677
Flickr	**0.091**	0.132	0.119	8.034
Deli-ui	0.646	0.636	**0.131**	96.703
DBLP	0.751	0.911	**0.487**	35.803
Syngraph1	1.101	6.293	**0.462**	5.506
Syngraph2	3.152	11.474	**0.402**	5.479

Table 10 Query time of *EETQ* (in seconds)

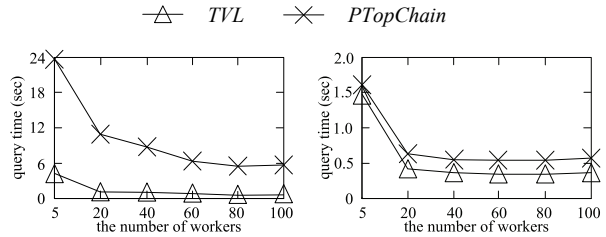
Graph	*TVL*	*PTopChain*	*GTopChain*	*Bi-BFS*
Wikipedia	**0.102**	0.440	0.246	12.463
Youtube	**0.027**	0.045	0.124	0.601
Flickr	**0.038**	0.117	0.122	6.597
Deli-ui	0.305	0.718	**0.188**	86.598
DBLP	**0.417**	0.702	0.465	23.728
Syngraph1	0.844	5.896	**0.598**	4.908
Syngraph2	3.024	9.883	**0.653**	5.187

In addition, the query latency for both positive and negative queries are reported in Table 7 and Table 8, respectively. We see that *TVL* outperforms *PTopChain* and *Bi-BFS* for both positive and negative queries on most datasets. This is because some reachable pairs of vertices are identified by Lemma 1. Some non-reachable pairs of vertices are eliminated by Lemma 2. Others that cannot be validated by Lemma 1 and Lemma 2 perform *BFS* efficiently since Lemma 3 helps shrink the search space.



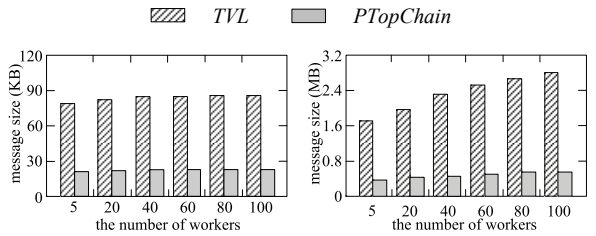
(a) *Youtube* (b) *Wikipedia*

Fig. 9 Query time vs. I



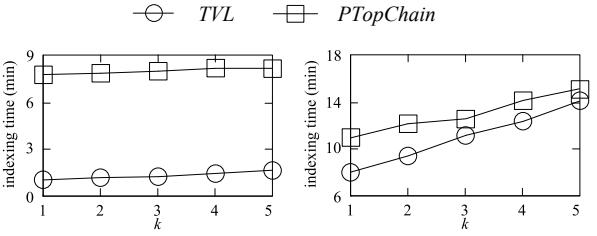
(a) *Syngraph1* (b) *DBLP*

Fig. 10 Query time vs. the number of workers



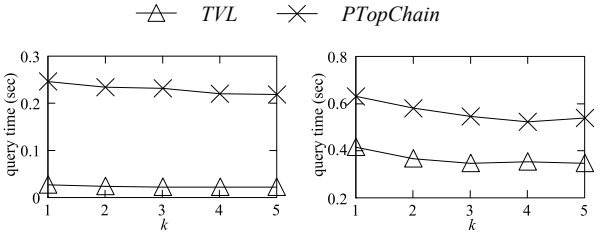
(a) *Syngraph1* (b) *DBLP*

Fig. 11 Communication cost vs. the number of workers



(a) *Wikipedia* (b) *DBLP*

Fig. 12 Index construction cost vs. k



(a) *Wikipedia* (b) *DBLP*

Fig. 13 Query time vs. k

Evaluating *MDQ* and *EETQ*. The second set of experiments investigates the query efficiency of *MDQ* and *EETQ*, with the results shown in Table 9 and Table 10, respectively. The query time of *DSR* and *disReach* is omitted since *DSR* and *disReach* have been shown to be very inefficient in the first set of experiments. We first observe that *TVL* achieves the highest efficiency on small-size datasets, while *GTopChain* achieves the highest efficiency on large-size datasets. The reason is the same as that analysed in *TRQ*. Next, we observe that *TVL* outperforms *Bi-BFS* on all datasets and outperforms *PTopChain* on 6 out of 7 datasets. For the remaining dataset, the query time of *TVL* is close to that of *PTopChain*. *PTopChain* outperforms *Bi-BFS* on 5 datasets. This is because *TVL* and *PTopChain* are able to effectively prune the search space. Last, we observe that for our proposed algorithms, compared with *TRQ*, both *MDQ* and *EETQ* take longer query time. The reason is that both *MDQ* and *EETQ* need to inspect all time-respecting paths between a source vertex s and a target vertex t within a time interval I to return a result. In contrast, *TRQ* stops once a time-respecting path between s and t within I is found.

Effect of time interval. The third set of experiments explores the impact of the input time interval $I = [w_s, w_a]$ on query performance. Following Wu et al. [41], we consider four different time intervals, I_i $(1 \leq i \leq 4)$, where I_1 is the maximal time interval of a dataset and $I_{i+1}(1 \leq i \leq 3)$ is the first subinterval after dividing I_i into two equal sub-intervals. Fig. 9 plots the results on *Youtube* and *Wikipedia*. It is observed that the query time of *TVL*, *PTopChain*, and *Bi-BFS* decreases as the interval shrinks. The reason behind this is that when time interval becomes shorter, queries can be answered in early supersteps via Lemmas 1 through 3 or *PTopChain*'s pruning strategies, resulting in reduced query costs. Again, *TVL* consistently outperforms the other methods.

Effect of the number of workers. The fourth set of experiments studies the impact of the number of workers on *TVL* and *PTopChain*. We vary the number of workers from 5 to 100, and report the query performance on *DBLP* and *Syngraph1* in Fig. 10 and Fig. 11, respectively. As expected, the query cost first drops and then stays stable or increases when the number of workers (denoted by $\sharp$*workers*) grows. The reason is that with more workers, there is more parallelism and thus more computational resources, but using more workers may also incur more communication overhead (as shown in Fig. 11(b)). At the beginning, the increased computational resources outweigh the increased com-

Table 11 Statistics of synthetic temporal graphs

Graph	$\|V\|$	E	$T(G)$	$\|\mathscr{V}\|$	$\|\mathscr{E}\|$
G_1	10,000,000	60,000,000	160	117,776,991	185,227,716
G_2	20,000,000	120,000,000	160	235,550,803	370,455,493
G_3	30,000,000	180,000,000	160	353,334,957	555,684,708
G_4	40,000,000	240,000,000	160	471,109,498	740,920,588
G_5	50,000,000	300,000,000	160	588,887,292	926,145,777

Table 12 Index size for *TVL*, *TopChain*, and *Grail* (in GB)

Methods	G_1	G_2	G_3	G_4	G_5
TVL	**0.79**	**1.59**	**2.39**	**3.19**	**3.99**
TopChain	3.73	7.45	11.18	14.90	—
Grail	1.76	3.51	—	—	—

Table 13 Indexing time for *TVL*, *TopChain*, and *Grail* (in seconds)

Methods	G_1	G_2	G_3	G_4	G_5
TVL	**61.68**	**128.04**	**198.34**	**232.61**	**276.58**
TopChain	223.82	590.31	3781.1	300072.2	—
Grail	4597.09	57883.332	—	—	—

munication cost, but then the increased overhead of inter-machine communication outweighs the increased computational capability. Again, *TVL* still excels over *PTopChain*. In addition, the query time reduced around two times by increasing $\sharp$*workers* from 5 to 20 on *DBLP*, indicating that the proposed algorithm benefits from the distributed system.

Effect of k. The fifth set of experiments considers the influence of parameter k on the indexing efficiency and query performance, using *Wikipedia* and *DBLP*. We change the value of k from 1 to 5 and report the results in Fig. 12 and Fig. 13. It is observed that, in Fig. 13, query costs of *TVL* and *PTopChain* first decrease and then tend to stabilize or increase slightly as k increases, which is due to more reachable information being maintained. Nevertheless, a larger k incurs higher index construction cost (as illustrated in Fig. 12) and does not improve query efficiency. Specifically, on *Wikipedia*, when k exceeds 4, the querying times of *TVL* and *PTopChain* do not decrease as k increases. On *DBLP*, when k exceeds 3, the querying time of *TVL* does not decrease with the growth of k; and the query performance of *PTopChain* does not improve when k exceeds 4. As a result, a smaller k is sufficient to enable the best query performance as well as index construction performance.

Scalability of *TVL*. The last set of experiments explores the scalability of *TVL* compared with *TopChain* and the representative centralized scalable method *Grail*, the source code of which was obtained from [47][6]. We generate five synthetic temporal graphs G_1, G_2, $\cdots$, G_5 of different sizes. Relevant statistics, including the number $|\mathscr{V}|$ of vertices and the number $|\mathscr{E}|$ of edges in transformed graphs $\mathscr{G}$ $(1 \leq i \leq 5)$ are summarized in Table 11. Specifically, we vary the number $|V|$ of vertices in these temporal graphs from 1×10^7 to 5×10^7, keep the number $|E|$ of edges at 6 times $|V|$, and fix the value of $|T(G)|$ at 160. Table 12, Table 13, and Table 14 plot the index size, indexing time, and query time of *DRQ* for *TVL*, *TopChain* and *Grail*, respectively.

Table 14 Query time of *TRQ* for *TVL*, *TopChain*, and *Grail* (in milliseconds)

Methods	G_1	G_2	G_3	G_4	G_5
TVL	568.88	692.92	908.64	1006.08	**1022.02**
TopChain	0.047	**0.114**	**18**	**168**	—
Grail	**0.015**	2.828	—	—	—

The first observation is that the index size of *TVL* is up to 4 times smaller than that of *TopChain*, and is 1.2 times smaller than that of *Grail*, as shown in Table 12. The second observation is that, as depicted in Table 13, the indexing cost of *TVL* is two orders of magnitude lower than that of *Grail* and *TopChain* on average, and *TVL* scales roughly linearly with the graph size. The third observation is that as plotted in Table 14, *Grail* is faster than *TVL* on G_1 and G_2; and *TopChain* is faster than *TVL* on G_1 to G_4. Nevertheless, *Grail* is unable to run on G_3, G_4, and G_5, and *TopChain* is unable to run on G_5. In contrast, *TVL* is able to run on all 5 graphs. The reason behind is that when the input graphs together with the indexes fit in the main memory of a machine and there is enough memory available to run *Grail* and *TopChain*, *Grail* and *TopChain* must be more efficient than *TVL* because there is no communication overhead or other overhead (such as barrier synchronization and task scheduling) of distributed processing for *Grail* and *TopChain*. Nevertheless, for large graphs, *Grail* and *TopChain* are unable to run because of memory overflow, while *TVL* works and is efficient. The result offers evidence that *TVL* is more scalable

[6] Code of *Grail* is available at `https://code.google.com/archive/p/grail/`.

Table 15 Answers for a case study (A = Tech Ridge Bay, B = GUADALUPE/30TH STREET)

Queries	$TRQ(A, B, [6\ p.m., 8\ p.m.])$	$EERQ(A, B, [6\ p.m., 8\ p.m.])$	$MDQ(A, B, [6\ p.m., 8\ p.m.])$
Answers	*true*	18:49:18	40.25 minutes

than *Grail* and *TopChain*.

Case study. we conduct a simple case study to assess the effectiveness of the three types of queries. A public dataset *Austin*[7], which records the timetable of the public transportation network of a major city *Austin* on a weekday, is used here. Each vertex in *Austin* represents a station, and each directed edge (u, v, s_t, a_t) is associated with a timetable $I = [s_t, a_t]$ that records the departure (resp. arrival) time of each vehicle at station u (resp. v). If a group of travellers are to depart from station A (= Tech Ridge Bay) no sooner than time s_t (= 6 p.m.) and arrive at station B (= GUADALUPE/30TH STREET) no later than time a_t (= 8 p.m.), consider three queries: (a) is there a route from station Tech Ridge Bay to station GUADALUPE/ 30TH STREET within [6 p.m., 8 p.m.]? (b) What is the earliest time they can arrive at station GUADALUPE/ 30TH STREET within [6 p.m., 8 p.m.]? (c) What is the minimal duration of their trip from Tech Ridge Bay to GUADALUPE/30TH STREET within [6 p.m., 8 p.m.]? To answer queries (a), (b), and (c), $TRQ(A,B,$[6 p.m., 8 p.m.]), $EERQ(A,B,$[6 p.m., 8 p.m.]) and $MDQ(A, B,$ [6 p.m., 8 p.m.]) are issued, respectively. Table 15 reports answers returned by the three queries. There is a route from Tech Ridge Bay to GUADALUPE/30TH STREET within [6 p.m.,8 p.m.], the earliest arrival time is 18 : 49 : 18, and the minimum duration is 40.25 minutes. This case study shows how reachability queries defined on temporal graphs make sense in real-life applications.

Summary. When faced with large temporal graphs, the centralized method *Grail* and *TopChain* cannot run. In contrast, on both medium-sized temporal graphs and large temporal graphs, the query performance of the *TVL* based methods is better than that of the *PTopChain* based approaches, and the index construction cost of *TVL* is lower than that of *PTopChain*. In addition, *TVL* can scale to very large graphs without significant loss of efficiency.

7 Conclusions

We propose an efficient index called Temporal Vertex Labeling (*TVL*), which is a labeling scheme designed for distributed temporal graphs. In addition, we present query algorithms that use *TVL* to support three types of reachability queries (i.e., temporal reachability query, earliest end time query, and minimum duration query) on temporal graphs. We provide three non-trivial lemmas that enable improved efficiency. An extensive experimental evaluation on seven real and synthetic data sets demonstrates that, compared with existing techniques, *TVL* is a scalable index that has low query cost and low index construction overhead. *TVL* also offers efficient support for dynamic insertion operations. In the future, it is of interest to investigate efficient distributed partitioning techniques for temporal graphs so as to further improve the performance of *TVL* based methods. It is also of interest to develop efficient indexes for supporting fast vertex/edge deletion and temporal interval update operations.

[7] *Austin* is available at `https://code.google.com/archive/p/googletransitdatafeed/wikis/PublicFeeds.wiki`.

8 Acknowledgments

This work was supported in part by the National Key R&D Program of China under Grant No. 2018YFB1004003, the NSFC under Grant No. 61972338, the NSFC-Zhejiang Joint Fund under Grant No. U1609217, and the ZJU-Hikvision Joint Project, and the National Research Foundation, Prime Minister's Office, Singapore under its International Research Centres in Singapore Funding Initiative. Yunjun Gao is the corresponding author of the work.

References

1. Agrawal, R., Borgida, A., Jagadish, H.V.: Efficient management of transitive relationships in large data and knowledge bases. In: SIGMOD, pp. 253–262 (1989)
2. Batarfi, O., Shawi, R.E., Fayoumi, A.G., Nouri, R., Beheshti, S., Barnawi, A., Sakr, S.: Large scale graph processing systems: Survey and an experimental evaluation. Cluster Computing **18**(3), 1189–1213 (2015)
3. Casteigts, A., Flocchini, P., Quattrociocchi, W., Santoro, N.: Time-varying graphs and dynamic networks. IJPEDS **27**(5), 387–408 (2012)
4. Chen, L., Gupta, A., Kurul, M.E.: Stack-based algorithms for pattern matching on dags. In: VLDB, pp. 493–504 (2005)
5. Chen, Y., Chen, Y.: An efficient algorithm for answering graph reachability queries. In: ICDE, pp. 893–902 (2008)
6. Cheng, J., Huang, S., Wu, H., Fu, A.W.: Tf-label: A topological-folding labeling scheme for reachability querying in a large graph. In: SIGMOD, pp. 193–204 (2013)

7. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
8. Fan, W., Wang, X., Wu, Y.: Performance guarantees for distributed reachability queries. PVLDB **5**(11), 1304–1315 (2012)
9. Gao, Y., Miao, X., Chen, G., Zheng, B., Cai, D., Cui, H.: On efficiently finding reverse k-nearest neighbors over uncertain graphs. VLDB J. **26**(4), 467–492 (2017)
10. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: Graphx: Graph processing in a distributed dataflow framework. In: OSDI, pp. 599–613 (2014)
11. Gurajada, S., Theobald, M.: Distributed set reachability. In: SIGMOD, pp. 1247–1261 (2016)
12. Holme, P., Saramäki, J.: Temporal networks. Physics reports **519**(3), 97–125 (2012)
13. Huang, S., Cheng, J., Wu, H.: Temporal graph traversals: Definitions, algorithms, and applications. CoRR **abs/1401.1919** (2014)
14. Huang, S., Fu, A.W., Liu, R.: Minimum spanning trees in temporal graphs. In: SIGMOD, pp. 419–430 (2015)
15. Jagadish, H.V.: A compression technique to materialize transitive closure. ACM Trans. Database Syst. **15**(4), 558–598 (1990)
16. Jin, R., Ruan, N., Dey, S., Yu, J.X.: SCARAB: scaling reachability computation on large graphs. In: SIGMOD, pp. 169–180 (2012)
17. Jin, R., Ruan, N., Xiang, Y., Wang, H.: Path-tree: An efficient reachability indexing scheme for large directed graphs. ACM Trans. Database Syst. **36**(1), 7:1–7:44 (2011)
18. Jin, R., Wang, G.: Simple, fast, and scalable reachability oracle. PVLDB **6**(14), 1978–1989 (2013)
19. Jin, R., Xiang, Y., Ruan, N., Wang, H.: Efficiently answering reachability queries on very large directed graphs. In: SIGMOD, pp. 595–608 (2008)
20. Kostakos, V.: Temporal graphs. Physica A: Statistical Mechanics and its Applications **388**(6), 1007–1023 (2009)
21. Koubarakis, M., Stamou, G.B., Stoilos, G., Horrocks, I., Kolaitis, P.G., Lausen, G., Weikum, G. (eds.): Reasoning Web. Reasoning on the Web in the Big Data Era, *Lecture Notes in Computer Science*, vol. 8714. Springer (2014)
22. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Distributed graphlab: A framework for machine learning in the cloud. PVLDB **5**(8), 716–727 (2012)
23. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: SIGMOD, pp. 135–146 (2010)
24. Michail, O., Spirakis, P.G.: Traveling salesman problems in temporal graphs. Theor. Comput. Sci. **634**, 1–23 (2016)
25. Nicosia, V., Tang, J.K., Musolesi, M., Russo, G., Mascolo, C., Latora, V.: Components in time-varying graphs. CoRR **abs/1106.2134** (2011)
26. Pan, R.K., Saramäki, J.: Path lengths, correlations, and centrality in temporal networks. CoRR **abs/1101.5913** (2011)
27. Redmond, U., Cunningham, P.: Temporal subgraph isomorphism. In: ASONAM, pp. 1451–1452 (2013)
28. Redmond, U., Cunningham, P.: Subgraph isomorphism in temporal networks. CoRR **abs/1605.02174** (2016)
29. van Schaik, S.J., de Moor, O.: A memory efficient reachability data structure through bit vector compression. In: SIGMOD, pp. 913–924 (2011)
30. Seufert, S., Anand, A., Bedathur, S.J., Weikum, G.: FERRARI: flexible and efficient reachability range assignment for graph indexing. In: ICDE, pp. 1009–1020 (2013)
31. Shao, B., Wang, H., Li, Y.: Trinity: A distributed graph engine on a memory cloud. In: SIGMOD, pp. 505–516 (2013)
32. Su, J., Zhu, Q., Wei, H., Yu, J.X.: Reachability querying: Can it be even faster? TKDE **29**(3), 683–697 (2017)
33. Tian, Y., Balmin, A., Corsten, S.A., Tatikonda, S., McPherson, J.: From "think like a vertex" to "think like a graph". PVLDB **7**(3), 193–204 (2013)
34. Trißl, S., Leser, U.: Fast and practical indexing and querying of very large graphs. In: SIGMOD, pp. 845–856 (2007)
35. Ueno, K., Suzumura, T., Maruyama, N., Fujisawa, K., Matsuoka, S.: Efficient breadth-first search on massively parallel and distributed-memory machines. Data Science and Engineering **2**(1), 22–35 (2017)
36. Wang, H., He, H., Yang, J., Yu, P.S., Yu, J.X.: Dual labeling: Answering graph reachability queries in constant time. In: ICDE, p. 75 (2006)
37. Wang, S., Lin, W., Yang, Y., Xiao, X., Zhou, S.: Efficient route planning on public transportation networks: A labelling approach. In: SIGMOD, pp. 967–982 (2015)
38. Wei, H., Yu, J.X., Lu, C., Jin, R.: Reachability querying: An independent permutation labeling approach. PVLDB **7**(12), 1191–1202 (2014)
39. Wu, H., Cheng, J., Huang, S., Ke, Y., Lu, Y., Xu, Y.: Path problems in temporal graphs. PVLDB **7**(9), 721–732 (2014)
40. Wu, H., Huang, Y., Cheng, J., Li, J., Ke, Y.: Efficient processing of reachability and time-based path queries in a temporal graph. CoRR **abs/1601.05909** (2016)
41. Wu, H., Huang, Y., Cheng, J., Li, J., Ke, Y.: Reachability and time-based path queries in temporal graphs. In: ICDE, pp. 145–156 (2016)
42. Yan, D., Cheng, J., Lu, Y., Ng, W.: Blogel: A block-centric framework for distributed computation on real-world graphs. PVLDB **7**(14), 1981–1992 (2014)
43. Yan, D., Cheng, J., Lu, Y., Ng, W.: Effective techniques for message reduction and load balancing in distributed graph computation. In: WWW, pp. 1307–1317 (2015)
44. Yan, D., Tian, Y., Cheng, J.: Systems for Big Graph Analytics. Springer Briefs in Computer Science. Springer (2017)
45. Yang, Y., Yan, D., Wu, H., Cheng, J., Zhou, S., Lui, J.C.S.: Diversified temporal subgraph pattern mining. In: SIGKDD, pp. 1965–1974 (2016)
46. Yano, Y., Akiba, T., Iwata, Y., Yoshida, Y.: Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In: CIKM, pp. 1601–1606 (2013)
47. Yildirim, H., Chaoji, V., Zaki, M.J.: GRAIL: a scalable index for reachability queries in very large graphs. VLDB J. **21**(4), 509–534 (2012)
48. Yildirim, H., Chaoji, V., Zaki, M.J.: DAGGER: A scalable index for reachability queries in large dynamic graphs. CoRR **abs/1301.0977** (2013)
49. Yu, J.X., Cheng, J.: Graph reachability queries: A survey. In: Managing and Mining Graph Data, pp. 181–215 (2010)
50. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: NSDI, pp. 15–28 (2012)

51. Zhang, X., Chen, L.: Distance-aware selective online query processing over large distributed graphs. Data Science and Engineering **2**(1), 2–21 (2017)

52. Zhu, A.D., Lin, W., Wang, S., Xiao, X.: Reachability queries on large dynamic graphs: A total order approach. In: SIGMOD, pp. 1323–1334 (2014)